Federal Office
for Information Security

Machine Learning in the Context of Static Application Security Testing - ML-SAST

Document history

Version	*Date*	*Editor*	*Description*
Preliminary	02.11.2021	team neusta, Uni HB	First Submission to BSI
First Update	15.11.2021	team neusta, Uni HB	Revision of Chapters
Final Version	24.11.2021	team neusta, Uni HB	Due Submission
Version 1.1	06.04.2022	team neusta, Uni HB	Minor Revisions
Version 1.2	25.11.2022	team neusta, Uni HB	Addendum Submission
Version 1.3	12.12.2022	team neusta, Uni HB	Revision of Addendum
Version 1.4	03.02.2023	team neusta, Uni HB	Revision of Addendum

Table 1: Version History

Federal Office for Information Security
P.O. Box 20 03 63
53133 Bonn
E-Mail: ki-kontakt@bsi.bund.de
Internet: https://www.bsi.bund.de

Authors and Acknowledgements

This study was executed by team neusta in cooperation with the University of Bremen on behalf of the Federal Office for Information Security (BSI). This publication was authored by:

Lorenz Hüther
Bernhard J. Berger
Stefan Edelkamp
Sebastian Eken
Lara Luhrmann
Hendrik Rothe
Marcus-Sebastian Schröder
Karsten Sohr

We want to thank our interview partners for giving insights into their work and business, on their thoughts and recommendations. The same thanks go to the people that filled the online questionnaire.

Table of Contents

1 Introduction 8
1.1 Summary 8
1.2 Summary in German 8
1.3 Motivation 8
1.4 Contribution 9
1.5 Structure 10
2 Artificial Intelligence and Machine Learning 11
2.1 Definition of Artificial Intelligence 11
2.2 Basic Principles of AI Systems 12
2.3 Machine Learning Primer 12
2.3.1 Supervised Learning 13
2.3.2 Unsupervised Learning 13
2.3.3 Reinforcement Learning 14
2.3.4 Neural Networks 14
2.3.5 Recurrent Neural Networks 16
2.3.6 LSTM Networks 17
2.3.7 Bidirectional Sequence Models 19
2.3.8 Graph Neural Networks 19
2.4 Other Concepts and Further Reading 20
3 Static Application Security Testing 22
3.1 SAST Tools 22
3.2 Background on Static Program Analysis 23
3.2.1 Compiler Construction Techniques 23
3.2.2 Typical Internal Data Structures for Static Program Analysis 24
3.2.3 Static Analysis Problems 26
4 ML in the Context of SAST 27
4.1 Preliminaries 27
4.2 Refinements 27
4.3 Current State of the Art 28
4.3.1 Datasets 28
4.3.2 Architectures 28
4.3.3 Explainability 28
4.3.4 Outlook 29
5 Expert Interviews 30
5.1 Method 30
5.2 Interview Partners 31

5.3 Results of the Expert Interviews 31
5.3.1 Versions of C and C++ Used 32
5.3.2 Opinions Towards SAST Tools in General 32
5.3.3 Input Format for Rules for SAST tools 33
5.3.4 Expectations Towards ML-based SAST Tools and Ideas to this End 33
5.3.5 Easily and Not Easily Detectable Errors 35
5.3.6 Sources for Error Use Cases 36
6 Online Questionnaire 37
6.1 Platform 37
6.2 Participants 37
6.3 Questionnaire 37
6.4 Results 38
6.4.1 C/C++ Versions 38
6.4.2 SAST Tools 39
6.4.3 Input Format for Rules for SAST tools 40
6.4.4 Error Use Cases 40
6.4.5 ML and SAST Tools 41
7 Elicitation and Explanation of Error Use Cases with Descriptions 42
7.1 void* Pointers 42
7.2 Difficulty in Tracking Tainted Data 43
7.3 Integer Overflows Related to Tainted Data 45
7.4 Array Boundary Errors 46
7.5 Null Pointer Access 47
7.6 Format String Errors (With Tainted Data) 48
7.7 String Termination Errors 49
7.8 Missing Security Checks for Critical Operations 49
7.9 Unsafe Use of Random Number Generators 50
7.10 Information Flow Through Uncleared Main Memory Data Structures 50
7.11 Insecure Origin of Cryptographic Keys and Initialization Vectors 50
7.12 Reuse of Initialization Vectors for Specific Operating Modes of Block Ciphers 52
7.13 Memory Leaks 52
7.14 Dangling Pointers – Use-After-Free 53
7.15 Race Conditions 54
7.16 Application-Specific Errors 54
7.17 Pointer Casts to Types with Stricter Alignment 55
8 Evaluation Criteria for ML-SAST Approaches 56
8.1 Qualitative Criteria 56
8.1.1 Transparency 56

8.1.2 Other Requirements 57
8.2 Quantitative Criteria 58
8.3 Other Criteria 61
9 Data in ML-Experiments: Types, Formats and Schemata 62
9.1 Benchmarks 62
9.2 Type, Format and Metrics 63
9.2.1 Name, Version and Description of Dataset 63
9.2.2 Dataset Scheme 63
9.2.3 Experiment Scheme 64
9.2.4 Model Persistence and Tracking 65
9.3 Ontologies for Machine Learning 65
10 Approaches to Conducting a Literature Mapping for ML-SAST 66
10.1 Research Questions 66
10.2 Initial Search 67
10.3 Iteration Steps 68
10.4 Data Extraction 68
10.5 Summary of the Most Important Results 69
11 Detailed Description of Most Appropriate Methods for ML-SAST 75
11.1 Training Data 75
11.2 Models 76
11.3 Explainability 78
12 Prioritization of Models for this Approach and Outlook on Further Work 82
12.1 Graph Neural Networks 82
12.2 Conventional Neural Networks 83
12.3 Clustering Approaches 83
12.4 Reinforcement Learning 85
12.5 Ensembles of Approaches 86
12.6 Conclusion of the Prioritization 86
13 Prototypical Implementation of a ML-SAST Tool 88
13.1 Presuppositions and underlying problems 88
13.1.1 Unsupervised Methods 88
13.1.2 Embeddings 89
13.1.3 Lack of Software 93
13.1.4 Summary 94
13.2 Development of the Prototype 95
13.2.1 Basic Approach 95
13.2.2 Demonstration Pipeline 98
13.2.3 Graph Generation Phase 99

13.2.4 Path Extraction Phase 100
13.2.5 Detection Phase 103
13.2.6 Front End 113
13.3 Evaluation 115
13.3.1 Juliet Test Suite Experiments 116
13.3.2 Oracle Tests 120
13.4 Limitations and Threats to Validity 123
13.5 Summary and Discussion 125
14 Conclusion 127
14.1 Further Research 127
14.2 Outlook 129
15 Appendix 131
15.1 Glossary 131
15.2 Results of the Literature Mapping 144
15.2.1 Base Set 144
15.2.2 Round I 146
15.2.3 Round II 147
15.2.4 Round III 148
15.2.5 Round IV 149
15.3 Guideline for the interviews with the experts 149
Bibliography 151

1 Introduction

This study begins with a short summary of its contents, both in English and in German. Then following, the motivation behind its conception, contribution and structure are presented.

1.1 Summary

In the research areas of formal methods and software engineering, various algorithms and techniques exist that perform static program analysis, e.g., to detect redundant code or programming rule violations via analyzing variable dependencies or removing possible flaws in device drivers. One more recent request is to find security issues in program code that can be exploited by an attacker, such as buffer overflows.

This study aims to scope and summarize the research area of machine learning-based approaches to static analysis security testing (SAST) so that subsequent efforts concerning the conception of a concrete tool may then resort to the results of this work. Such a tool could, for instance, exist as an addition to the compiler, which warns programmers of producing insecure code that may enable malicious attacks. Machine learning algorithms based on textual pattern detection and abstract syntax tree analyses or distinguished features are applied here. Such detection technique does not have to suffice the requirement of providing perfect detection rates but instead should provide a lower false positive ratio than conventional methods to offer a timely assessment during the development process of code.

1.2 Summary in German

Im Bereich formaler Methoden und Softwareentwicklung finden sich verschiedenste Algorithmen und Techniken, die eine statische Analyse des Quelltextes vornehmen, z.B. um redundanten Code, Verletzungen von Programmierrichtlinien, oder Fehler in Treiberdateien aufzudecken. Ein aktuelles Anliegen ist es, bereits während der Programmierung Sicherheitslücken für verdeckte Angriffsmöglichkeiten, wie Pufferüberläufe, aufzudecken.

Diese Studie zielt auf ein Tool, das den Programmierer vor sicherheitskritischen Programmfehlern im Quellcode warnt. Zu diesem Zweck sollen maschinelle Lernverfahren basierend auf textueller Mustererkennung, abstrakten Syntaxbäumen oder ausgewählten Eigenschaftsvektoren eingesetzt werden. Ein solches Tool muss dabei nicht dem Anspruch genügen, eine perfekte Erkennungsrate vorweisen zu können. Vielmehr gilt es, eine gegenüber herkömmlichen Tools, reduzierte Fehlerrate zu erreichen, um eine zeitige Einschätzung während der Entwicklung von Software zu bieten.

1.3 Motivation

Static analysis is the automated analysis of source code without executing the application. When the analysis is performed during program execution, it is referred to as **dynamic analysis**. Static analysis is often used to detect security vulnerabilities, performance issues, non-compliance with standards or the use of out-of-date programming constructs.

The basic concept common to all static analysis tools is searching source code to identify specific coding patterns that have some sort of warning or information associated with them. However, static analysis tools vary in how they implement this functionality, e.g., source code parsing technology to create an **abstract syntax tree (AST)**, **text regular expression matching**, or a combination of the two.

Regular expression matching on text is very flexible, and it is easy to write rules to match. On the downside, however, such simple rules often lead to a high number of false positives, and the matching rules do not consider the surrounding code context. This holds in particular true, as purely regular languages cannot capture the complete semantics of the most likely Turing-complete languages they are checking. More advanced methods may rely on complex forms of code representation. AST matching, for instance, compiles the source code into a tree-like data structure prior to checking instead of relying on simple files filled with text. This allows for more specific, contextual matching and can reduce the number of false positives reported against the code. There are several other data structures to represent source code, subject to a static analysis. Even with more advanced forms of code representation or by the employment of more capable languages for the definition of rules, ultimately, one cannot deterministically reason about any non-trivial property of a program, as stated by Rice's theorem[1]. Consequently, whether a given program is free of vulnerabilities cannot be ascertained, as it is only possible to prove the opposite if a vulnerability is found. Doing so, however, poses a challenging task, and conventional rule-based methods are hindered by severe limitations, such as the ones stated above. The study aims to explore the possibility of applying machine learning techniques to this end to alleviate some of these limitations. Such techniques may be more capable, picking up on more subtle code patterns indicative of a vulnerability than conventional methods.

Static application security testing (SAST) is an integral part of a **security development lifecycle (SDL)** to detect security bugs in program code as early as possible. SAST tools are an important means to improve the security and quality of C/C++ program code as these programming languages allow low-level programming, which is error-prone[2], [3]. SAST tools promise to automatically detect low-level programming bugs, such as buffer overflows, heap overflows, race conditions or memory management errors, and over the last years, there has been great progress in this area. Despite some early progress, however, SAST tools may miss important security bugs, so-called **false negatives**, or vice versa, they may also produce **false positives**. While the former may lead to overlooked security holes, the latter could potentially fatigue the security analyst, who finally may ignore critical errors or even turn off the tool in certain situations.

1.4 Contribution

Due to these shortcomings, the question arises of how the functionality of SAST tools can be improved to obtain more precise results with better false positive and false negative rates. One possible way of enhancement is to use **machine learning (ML)** for static analysis in an **ML-SAST** approach. For example, ML may help to automatically infer rules indicative of vulnerable code patterns, which a conventional SAST tool could then use to detect errors in previously unseen code. In addition, ML may help a SAST tool to automatically learn insecure programming practices, such that unknown bugs can be detected earlier and with fewer false negative rates.

In recent years, several approaches to ML-SAST have been proposed in academia and by commercial tool vendors. To better understand the underlying technical concepts, both with respect to static analysis and ML methods, the prospects as well as the limitations of ML-SAST, a broad survey on current ML-SAST approaches is valuable for SAST-tool developers, researchers, and software security professionals.

The study addresses this demand in several ways. First, expert interviews with software security professionals are conducted. This serves the purpose of obtaining information on the limitations of current SAST tools, error use cases that are usually hard to detect by conventional SAST tools as well as possible approaches how such tools could be enhanced through machine learning technologies. Secondly, an online survey is conducted, which complements the interviews and helps one obtain a broader overview of the topic. Thirdly, the state-of-the-art in recent publications is explored through a systematic in-depth literature mapping with respect to ML-SAST. Its results are then discussed in depth, highlighting and reviewing promising approaches to this end. It is accompanied by a discussion of security error use cases, ML methods, and metrics applicable to SAST. The entire study was carried out over a period of about half a year.

This study's primary purpose is to provide the basis for the future development of a (public domain) ML-SAST tool. The audience of this study includes but is not limited to software security professionals, developers of SAST tools, and researchers interested in software security with a focus on C/C++ code.

1.5 Structure

The remainder of this study is organized as follows. Chapters 2-3 give an overview of the topics artificial intelligence (AI), ML, and SAST. After that, in Chapter 4, a brief overview over the historical development regarding ML-SAST is provided, highlighting noteworthy milestones in this area of research. The subsequent Chapters 5-6 then first document the expert interviews conducted as part of this study and thereafter the related online questionnaire. After describing interesting error use cases for C/C++, which ML-SAST tools should support, in Chapter 7, qualitative as well as quantitative requirements for the application of machine learning techniques for SAST are discussed in Chapter 8. Closely related, the following Chapter 9 then delves into the data present in the context of machine learning experiments, interchange formats and ontologies that ought to be considered during the development of ML-SAST tools in order to facilitate the satisfaction of the aforementioned requirements. Chapter 10 then provides extensive documentation of the literature mapping, conducted as part of this study, discussing the methodology to this end and the ensuing results. Chapters 11 and 12 describe the most promising and mature ML-SAST approaches in detail and aim to provide some guidance concerning a prioritization for the development of a concrete ML-based SAST tool. Finally, in Chapter 14, the study concludes with a summary and an outlook on further research options.

2 Artificial Intelligence and Machine Learning

This chapter poses as a basic introduction to **artificial intelligence (AI)** and **machine learning (ML)**, a subdomain of the former. In the following, first, the terms will be defined so as to provide the necessary context for the remainder of the study. Then the basic principles and technologies, behind these concepts will be elaborated on in detail. Here, especially neural networks play a central role, being the predominant technology in terms of machine learning based SAST approaches in the literature.

2.1 Definition of Artificial Intelligence

The conception of an overarching definition as to what artificial intelligence is, poses a challenging task. The BSI bases their definition of the term on that of the High-Level Expert Group on Artificial Intelligence (AI-HLEG), as:

> *"[...] the technology and scientific discipline that includes several approaches and techniques, such as machine learning, machine reasoning and robotics."[4]*

This study focuses particularly on the subdomain of machine learning, as already suggested by the abbreviation ML-SAST. This, however, immediately prompts the question of what machine learning is. The AI-HLEG describes machine learning techniques as follows:

> *"These techniques allow an AI system to learn how to solve problems that cannot be precisely specified, or whose solution method cannot be described by symbolic reasoning rules."[5]*

This is also the case for ML-SAST. It is not possible to efficiently describe vulnerabilities in the form of symbolic reasoning rules. Some vulnerable code patterns may be easier to express in this way than others. In general, however, such patterns are very nuanced, and the slightest changes to the code can decide between a correctly functioning program or a severe vulnerability. It is here, where the application of machine learning for the purpose of static code analysis may be of great potential. As conventional SAST tools rely on the manual definition of such rules, an ML-based approach may be able to recognize these subtle nuances that human practitioners have either difficulty to describe in a formal manner or recognize in the first place.

The technologies discussed in this study, encompass supervised as well as unsupervised machine learning algorithms, which do not rely on explicit formal definitions to solve a problem directly. Instead, these algorithms may be used to train some model, based on large sets of datapoints, representative of the underlying problem. These datapoints may be present in the form of code sections that may or may not contain a vulnerability. In case of supervised learning, the aforementioned datapoints are tuples of the code sections and an accompanying label that states whether or not this section is vulnerable. With such a dataset, it is then possible to employ machine learning algorithms to iteratively and automatically adjust a model that effectively approximates a solution to the problem of identifying vulnerabilities in source code.

Moreover, an unsupervised algorithm can be employed in cases where again a large dataset of code sections exists. This time, however, it is not known whether these sections are vulnerable or not. Here, such unsupervised algorithms can be used to automatically find clusters within the dataset in accordance to some pattern. Clustering the data in this way, vulnerable code patterns may then become observable in the form of outliers.

This is just a brief introduction to machine learning for the purpose of static code analysis. In the ensuing sections of this chapter, the aforementioned machine learning algorithms are elaborated on in more detail. On this note, specifically the supervised methods will be thoroughly covered, as a good understanding of their working principles is necessary for the chapters following. This is due to the fact that as of today supervised ML-SAST approaches pose the technology most often employed in the literature.

2.2 Basic Principles of AI Systems

At the BSI, three essential points of contact between AI and IT security are currently the subject of intensive research[4].

IT security for AI: How can AI systems be attacked? Which attack scenarios arise using AI and what effects do they have? How can AI systems be verifiably protected against AI-specific attacks?

IT security through AI: How can AI systems improve IT security? Which defense measures can be designed more efficiently, which can be newly developed?

Attacks by AI: What new threats to IT systems do AI methods create in attack tools? Which known attacks can AI improve? How can such attacks be recognized and be defended against?

This study is mainly concerned with the application of AI for security purposes. While attacks by AI are not covered, the attacks on AI do indeed pose a threat, albeit a small one, to the security of ML-based SAST solutions. In this regard, it would be conceivable, for instance, that an adversarial user may try to trick such solution into classifying a severe vulnerability as benign, which then undermines some given security goals. These topics are touched upon in Chapters 8 and 12.

In the following, three principles to which AI systems should adhere and which may be considered essential in the context of this study, are being discussed. Completeness is not claimed as there are many others noteworthy principles that have been published by the EU, NATO, USA/Whitehouse[6]–[8].

Reliability: AI-based systems must be able to operate reliably, securely, and continuously, and not only under normal conditions, but also under unexpected circumstances, or when they are attacked. Rigorous testing during development and implementation is critical. People should always play a pivotal role in making decisions about how and when to use AI systems.

Privacy: Hosted AI systems that process and possibly store personal information must comply with applicable data protection laws. These require transparency in the collection, processing and storage of data and provide suitable control mechanisms for consumers so that they can decide for themselves how their data is used. AI systems may only use private data in accordance with applicable data protection standards and must protect data securely against theft.

Transparency: AI will more and more affect people's lives. One should provide contextual information about how AI systems work so that people can understand AI decisions and recognize possible wrong decisions and prejudices more easily. This requirement is covered in detail in Chapter 8.

The demands on an AI system are high. In the context of this study, reliability is important, while privacy is a term that is applicable not only to AI systems, and transparency often is too high of a requirement. Self-explanatory AI systems are difficult to design. How can a deep neural network that has thousands and thousands of trained weight vectors and represents a complex graph-based function as a black box explain why it recognizes what and how? Chapter 8 elaborates on these concepts in much more detail.

2.3 Machine Learning Primer

Machine learning (ML) is the field of study that gives computers the ability to learn without being explicitly programmed. ML is one of the areas in AI that has made a significant progress in the last decade, which, in turn, has raised a renewed interest in the application of AI in general. Historically, even before the uprising of AI, ML has already been identified as a relevant research area, e.g., by Alan Turing in his seminal paper of Computing Machinery and Intelligence published in the journal Mind in 1950[9].

According to the High-Level Expert Group on Artificial Intelligence, ML includes neural networks, deep learning, decision trees, and many other learning techniques[5]. These techniques allow an AI system to learn how to solve problems that cannot be precisely specified, or whose solution method cannot be described by symbolic reasoning rules. Examples of such problems are those that have to do with perception capabilities such as speech and language understanding, as well as computer vision or behavior prediction. However, machine learning techniques can be used for many more tasks than only perception. Machine learning techniques produce a numeric model (that is, a mathematical formula) used to compute the decision from the data.

While computers are very good at solving mathematical problems that can be formally described, they are not very good at solving problems that cannot easily be expressed in a formal manner. With machine learning, instead of explicitly stating an algorithm to solve a specific problem directly, the machine is merely given the means to approximate the solution to a problem itself. In essence, there are three main types of machine learning: supervised learning, unsupervised learning, and reinforcement learning. These types are discussed in more detail subsequently

2.3.1 Supervised Learning

Supervised learning is characterized by the employment of datasets for training and testing. Both of which are collections of samples $\{(x_i, y_i)\}_{i=1}^{N}$, where $x_i$is a is a D-dimensional vector of features and a y_ia corresponding target[10, p. 3]. Such vector x could, for example, contain the color data of an image, whereas y could be used to describe the images content, i.e., a label. Essentially these datasets are subsets of the domain and their associated elements are from the codomain of the function that is being approximated, meaning that with supervised learning there is a **known target**.

Given a set of training data, the algorithm tries to optimize the parameters of a model in such a way that it closely resembles an unknown function, representative of the relationship between the input samples and targets[10, p. 6]. This function may then be used to map previously unseen input data x to a predicted output $\hat{y}$. When the prediction is a class, also referred to as category, the underlying problem is referred to as **classification**. In other cases, when the prediction is a number, it is called **regression**. Using the test data, it is possible to additionally assess the ability of the model to generalize from the known samples.

This method of training a model prior to making predictions is referred to as **model-based** learning. A concrete example of such an algorithm would be a support vector machine. **Instance-based** learning algorithms on the other hand do not rely on the prior training of a model, but instead use the dataset itself as the model. An example of this would be the k-Nearest-Neighbor algorithm. Here, given an input x, the algorithm considers known data points in the vector space that are in close vicinity to x, i.e., its neighborhood. It is then inferred that the label seen most often in that neighborhood must also apply to the unknown input x[10, p. 13].

Representatives for classification algorithms are, for instance, **decision trees (DTs)**, **naive Bayes (NB)**, **k-nearest neighbor (KNN)**, **linear classifier (LC)** or **support vector machines (SVMs)**:

- In DTs nodes are attributes, arcs are labelled with possible answers, and terminal nodes represent class labels, in NB the simplest probabilistic belief network is used,
- in KNN search the k closest entries in a case base use a majority vote for determining the class,
- LC computes a hyperplane in Euclidean space to separate the classes,
- SVM is a refinement of such binary classifier with support vectors on the margins that decide on the separating hyperplane. Mathematically speaking, a different loss function is applied.

2.3.2 Unsupervised Learning

Concerning **unsupervised learning**, there is a collection of unlabeled feature vectors $\{x_i\}_{i=1}^{N}$, without an explicit learning target definition. Unsupervised learning may be used, for instance, for clustering, where it is the goal to divide the said collection of unlabeled data into clusters that show some sort of similarity. **Hierarchical Bottom-up** and **K-means** are two examples of such clustering methods.

- Bottom-up clustering iteratively constructs a similarity tree, called dendrogram, while

- k-means is a fast, iterative procedure to compute cluster midpoints, called centroids, minimizing intra- and maximizing inter-cluster distance.

The opposite of clustering would be outlier detection. Here, the target is not to group similar data, but data that is different from the rest. Other examples of unsupervised learning comprise association rule learning or dimensionality reduction. Broadly speaking, unsupervised learning pursues the goal of deriving "interesting structures" from data. Thus, clustering approaches offer the benefit of not requiring the targets to be known beforehand, i.e., they do not require labeled training data. In some domains generating such training data may not pose much of a difficulty, whereas in other domains this task could be challenging.

2.3.3 Reinforcement Learning

In **reinforcement learning** the training dataset may not be fixed at the time of training, but instead the algorithm exists in an environment, the state of which it perceives as a vector of **features**[10, p. 4]. Not only does the algorithm perceive the aforementioned environment, but it also interacts with it by executing actions that then result in some reward or punishment. By trying to maximize the reward, the algorithm continuously adapts its policy[10, p. 4]. Function approximation, optimization problems and learning in multi-agent systems are some of the fields, where reinforcement learning may be applied. Representative algorithms are, for instance, **value iteration** and **policy iteration** or **Monte-Carlo search**.

- Value and policy iteration are deterministic procedures that apply a local update equation in a graph, or apply a solver to a linear set of equations[11, pp. 716–719].
- Monte-Carlo search is a recent class of randomized algorithms that is based on random rollouts, and a mechanism to trade exploitation with exploration[12, pp. 282–293].

2.3.4 Neural Networks

Neural networks are probably the most influential approach in machine learning and according to some publications identified in the literature mapping, better suited for SAST applications than other types of machine learning approaches, see Chapters 11 and 12. As such, this section provides an in-depth look at the underlying principles, by which artificial neural networks operate.

Neural networks are inspired by Biology. Real neurons have many interconnections, inputs, and outputs, and make use of simple thresholds. **Artificial neural networks** on the other hand, consist of several simple nodes that, in reference to their biological counterpart, are sometimes referred to as neurons, as well. Additionally, artificial neural networks have weighted edges that connect the individual nodes. A node may have several inputs, through which it receives the output of other nodes or direct input from some data sample, in case the node is part of the first layer of the network. In addition to the weights of the edges, the nodes themselves possess biases. The output of a node depends on the values present at its inputs and an activation function, i.e., a function that maps a real valued input to some real valued output. Different activation functions have different characteristics and the choice of which one to use is highly task specific. Figure 1 depicts an example of such a node with the inputs $x_1, x_2, \dots, x_n$ and their associated weights $w_1, w_2, \dots, w_n$. The weighted sum of all inputs and some bias b is calculated and subsequently transformed by an arbitrary activation function f(y).

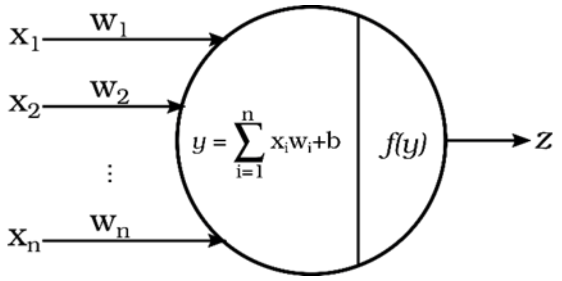


Figure 1 Exemplary depiction of a node with the weighted inputs and some activation function

A neural network learns through repeated adjustments of these weights and biases, in a process called **backpropagation**. These adjustments can be approximated by means of a **cost function** and some **optimization algorithm**. In the case of artificial neural networks, most often some form of **(stochastic) gradient descent** is used. The derivative of a function f(x) gives us the slope at the point x, denoted as f'(x). Gradients are much like derivatives, but for multi-variable functions. More formally, a functions gradient is a vector of **partial derivatives**, usually denoted as ∇f [10, p. 8]. As mentioned before, it is the goal to adjust the weights and biases of the neural network, in accordance with a set of known datapoints i.e., the training data. This can be achieved by letting the network make a prediction to some x from the samples given by $\{(x_i, y_i)\}_{i=1}^{N}$, denoted as $\hat{y}$. Since the actual value of y is known, it can be calculated by how much the prediction $\hat{y}$ deviates from the expected value y using the cost function. An example of such a cost function is the **mean squared error** or MSE for short, depicted in *Equation* 1.

$$\mathrm{MSE} = \frac{1}{n}\sum_{i=1}^{n}\left(Y_i - \hat{Y}_i\right)^2$$

Equation 1 *The mean squared error function or MSE for short, where* $\hat{Y}$ *is the predicted and* Y *the actual value.*

To estimate how the weights of the network need adjustment and to reduce the loss given by the cost function, the gradient of that cost function is computed. As the gradient shows the direction of the steepest ascent for a function, the negative gradient naturally shows the direction of steepest decline. Following the direction of steepest decline and adjusting the network's weights accordingly, it is possible to converge towards a local minimum of that function, hence reducing the loss. An exemplary depiction of the gradient descent can be seen in Figure 2.

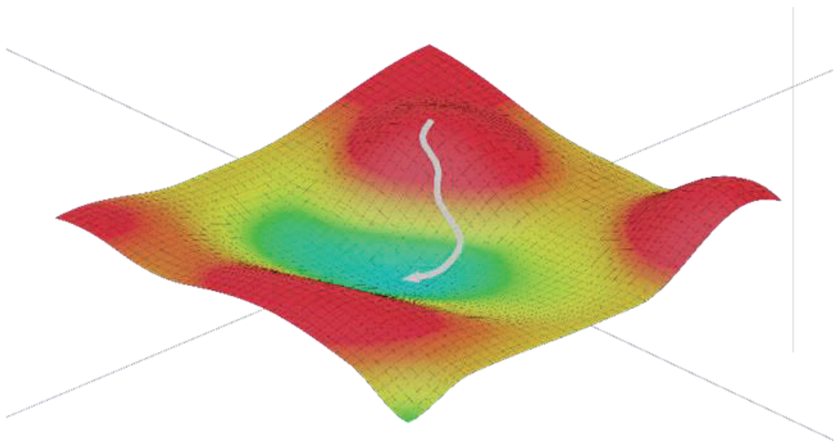


Figure 2 Exemplary depiction of the gradient descent for some two-dimensional function.

A neural network can have one or more layers (that is, single-layer and multi-layer neural networks). Each layer has one or more nodes. The **input layer** provides information from the outside world to the network. The output layer later transfers information to the outside world. The middle layers are called **hidden layers** because they have no direct connection with the outside world. Neural networks can have zero or more hidden layers. In **feed-forward networks** the information moves in only one direction, forward, from the input nodes, through the hidden nodes (if any) and to the output nodes. The nodes of recurrent neural networks have feedback loops. Usually, networks are fully meshed in between the layers. In **convolutional neural networks**, the mesh is only between some selected parts of nodes in the layers. In exchange, convolutional neural networks are often considered **deep**, in reference to their many hidden layers. A **perceptron** is the simplest neural network structure. It is a single-layer feed-forward neural network, and typically used for binary classification tasks. Portrayed in Figure 3 is a very simple three-layered feedforward neural network. Its input is a four-dimensional feature vector, and its output is a binary prediction. In-between the input and output layers, is one fully-connected hidden layer.

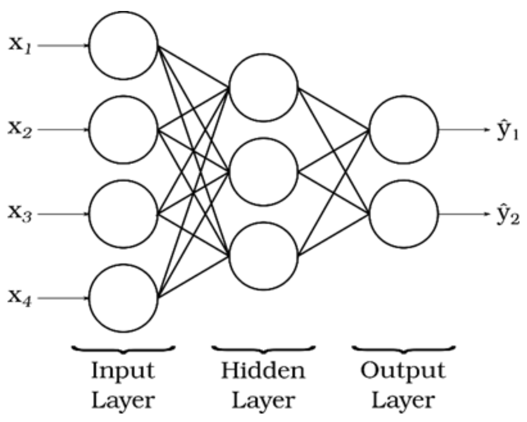


Figure 3 Simplified example of a three-layered artificial neural network.

2.3.5 Recurrent Neural Networks

There are many cases in which data may be portrayed as a sequence of individual datapoints. Natural language, for instance, is an example of such sequential data. Here, characters form words, words form sentences and sentences form complete texts. It is undeniable that the position of the individual elements, e.g., words in a sentence, is of great importance. Since natural languages follow some explicit grammar, rearranging the words of a sentence may result in a completely different meaning or incomprehensible gibberish. Moreover, the meaning of a single word also highly depends on its surrounding words, i.e., its context. As this example shows, not only is there information contained in the individual elements of such sequential data, but also in the structure itself.

Simple feed forward neural networks, such as multilayer perceptrons, are well suited for processing static data, such as images. As described however, a lot of problems involve **dynamic data**, which simple feed forward architectures fail to adequately address, as they lack the means to consider temporal correlations[13]. One popular approach that aims to solve this adds **feedback loops** to the network, such that the output $\hat{y}_t$ is not only computed based on the mere input, but also some hidden state h_t that is being updated based on the previous state h_{t-1} and some input x_t. Figure 4 depicts a single module of a RNN[14, p. 2]. This modular depiction is often used to describe reusable units that may occur multiple times within a single network. Thus, a neural network may be viewed as a series of modules that themselves comprise one or more layers of nodes.

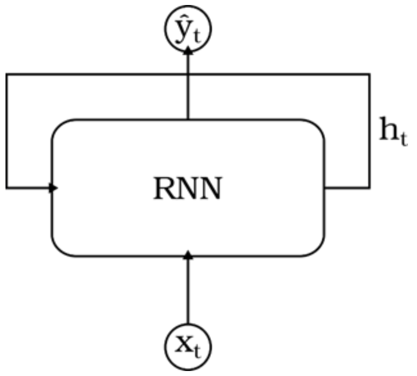


Figure 4 Simplified depiction of a recurrent neural network as a module, which itself may comprise multiple layers of nodes.

Using RNNs, different types of problems may be addressed. If for each time step t a prediction is made, it is possible to solve **many-to-many** problems, i.e., problems that involve finding some sequence, given another sequence. That is, since sequences may be regarded to as a series of discrete time steps as well, that may, one by one, be passed to the network as input. Then for each input step, an output can be predicted that itself forms sequence of discrete steps in time. It is also possible, however, to solve **many-to-one problems**, for example, sentiment analyses or **one-to-many problems** like generating a caption, given an image[15]. In an RNN, with each step t, the hidden state h_t is computed using a tanh-layer over the weighted previous state Wh_{t-1}, the weighted input Ux_t for matrices U,W and some bias b, thus yielding $h_t = \tanh(Wh_{t-1} + Ux_t + b)$. Moreover, the output $\hat{y}$ at step t is given by the product of the hidden state h_t and another weight matrix V, plus some other bias c, resulting in $\hat{y}_t = Vh_t + c$. Figure 5 shows a single module of an RNN that has been unrolled over its time steps[14, p. 2], [16, p. 546]. It must be noted that the layers of the network itself are not shown, but rather the flow of data over time at the same module.

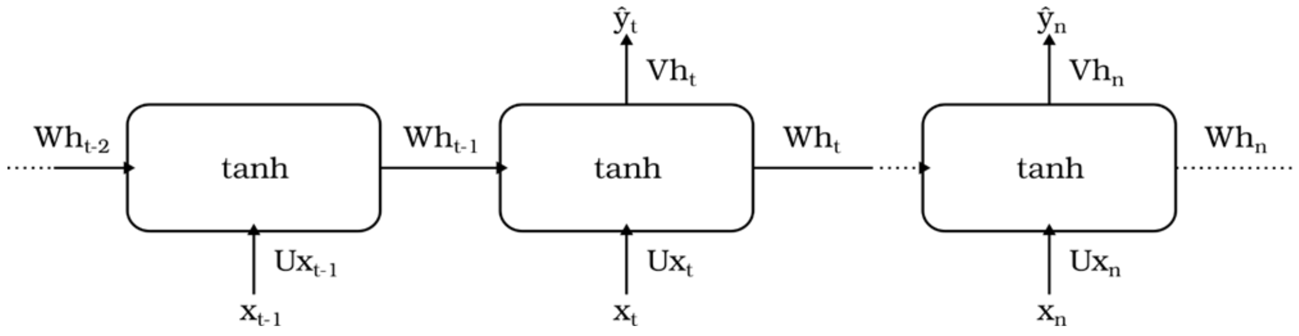


Figure 5 Depiction of a recurrent neural network as a chain of modules that has been unrolled over time.

Recurrent neural networks can be very forgetful, due to the fact that they use a modified version of the backpropagation technique, called **backpropagation through time**. This is necessary as with RNNs not only do the models coefficients need to be adjusted over all input samples, but also over all time steps. The loss is given by the sum of all losses L_t over all time steps $t \in T$, used for gradient calculation. This gradient also needs to be calculated for the weight matrix Wh_t that contains hidden state h_t at time step t. Unfortunately, however this involves derivatives $\frac{\partial h_t}{\partial h_{t-1}}$, which tend to decrease the gradient for a larger T if Wh has eigenvalues $|\lambda|^{T-1} < 1$. This problem is often referred to as the **vanishing gradient problem**. Vice versa, in the case that $|\lambda|^{T-1} > 1$ holds true, the gradient can grow infinitely large. This is known as the **exploding gradient problem**. Both cases have the effect that RNNs cannot be trained over long distances[16, pp. 550–554].

2.3.6 LSTM Networks

As laid out before, RNNs suffer from a tendency to lose their effectiveness in tasks involving causalities over long sequences. **Long Short-Term Memory (LSTM)** networks offer a solution to the vanishing gradient problem, but do not alleviate the effects of the exploding gradient problem. This type of network was first introduced by Hochreiter and Schmidhuber in 1997[16, p. 554], [17]. LSTM networks, belong to the class of recurrent neural networks and are thus tailored towards solving sequence prediction problems. This ability makes them well suited for complex problem domains like machine translation or speech recognition. As such they are not unlike conventional RNNs. The difference is that LSTM networks introduce the concept of cell state, denoted as c_t at time step t.

For regular RNNs such module could for instance be a tanh-layer, that computes an output $\hat{y}_t$ in accordance to some input x_t and the previous hidden state h_{t-1}. In the case of LSTM networks there are much more complicated modules involved, next to others, that make up for its architecture. Here, a single module comprises not just one, but four layers, each of which serving the purpose of manipulating the cell state c_t in accordance to the previous cell state c_{t-1}, the previous hidden state h_{t-1} and some input x_t. In the literature, these layers are often referred to as "gates" that control the flow of information through the network. First, the forget gate f, a sigmoid-layer, removes irrelevant information from h_{t-1} and multiplies it with c_{t-1}. Next, the update gate u, first computes a scale factor i_t through another sigmoid layer and secondly a so-called candidate matrix $\tilde{C}_t$ using a tanh-layer, again in accordance to h_{t-1} and x_{t-1}. Finally, the product $i_t\tilde{C}_t$ is added to the result of the forget operation, conducted prior. The last gate in the series, is the output gate o, realized by another sigmoid-layer over h_{t-1} and x_{t-1}. Next, the tanh function is applied to the cell state and multiplied with the output of the sigmoid layer, yielding the output h_t. I must be highlighted that the tanh operation applied to the cell state in the last step is a single operation and not an entire layer of individual nodes. It serves the purpose of ensuring that the state takes a value between -1 and 1.

Moreover, it is easily visible that this architecture allows for the cell state to pass unhindered between the modules, such that it is not subject to the vanishing gradient problem[16, pp. 550–555]. Figure 6 shows the structure of such LSTM module in a conceptual manner, depicting the gates discussed above and Equation 2 shows a comprehensive list of all the operations discussed above[18].

$$i_t = \sigma\left(x_t U^i + h_{t-1} W^i\right)$$
$$f_t = \sigma\left(x_t U^f + h_{t-1} W^i\right)$$
$$o_t = \sigma(x_t U^o + h_{t-1} W^o)$$
$$\tilde{C}_t = \tanh(x_t U^g + h_{t-1} W^g)$$
$$C_t = f_t \times C_{t-1} + i_t \times \tilde{C}_t$$
$$h_t = \tanh(C_t) \times o_t$$

Equation 2 A comprehensive list of all “gate”-operations performed within a single LSTM module. It must be noted that here, W and U denote the weight matrices for the module’s layers, which have been omitted in the description above for better understanding of the operations themselves.

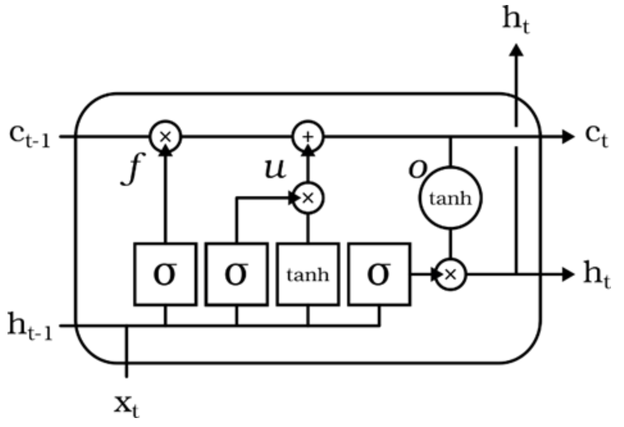


Figure 6 A LSTM module where c_t denotes the cell state, h_t the hidden state and x_t the input at time point t. The operations × being pointwise multiplication and + the sum over all inputs. astly σ and tanh represent the sigmoid and hyperbolic tangens activation functions and the letters f, u and o mark the conceptual location of the forget, update and output gates.

A concept similar to LSTM networks are **Gated Recurrent Unit (GRU)** networks. Much like LSTMs, these networks also make use of gates to selectively convey information through time. In the broadest sense, GRU networks pose an enhancement to conventional LSTMs as they are easier to implement and compute. Instead of having three gates, GRU networks only possess an update gate z that decides whether or not to update the current hidden state h_{t-1} with the new state h_t and a reset gate r that can selectively ignore the previous hidden state h_{t-1}. Moreover, GRU networks abandon the idea of cell state and instead rely on the hidden state only, much like the classic RNNs introduced in Section 2.3.5. Figure 7 depicts this simplified hidden activation function[19].

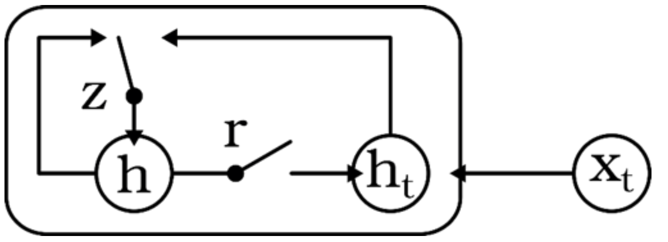


Figure 7 The hidden activation function of gated recurrent unit networks.

2.3.7 Bidirectional Sequence Models

The conventional versions of the sequence models discussed, i.e., RNN, LSTM and GRU networks all process the input in a single direction. Subsequently, such models can also only learn sequential dependencies in one single direction. At the example of ML-SAST for instance, a model could learn that the declaration of a pointer variable called `buffer` or `buf`, is often followed by a call to `malloc`. Bidirectional models, such as BRNN, BLSTM or BGRU networks on the other hand are capable of inferring sequential relationships in both, forward and backward direction. Schuster and Paliwal introduced the idea in 1997 with their proposal of bidirectional recurrent neural networks, by combining two unidirectional RNNs in a single architecture. The authors split the networks into two parts that are connected to the same input layer, but independently process the input in opposing directions. Only at the final output layer, the output of both sets of nodes is connected. This allows the network to also consider future information from the sequence, unavailable to conventional RNNs. According to Schuster and Paliwal this leads to improved predictive performance of BRNNs, compared to regular RNNs[20].

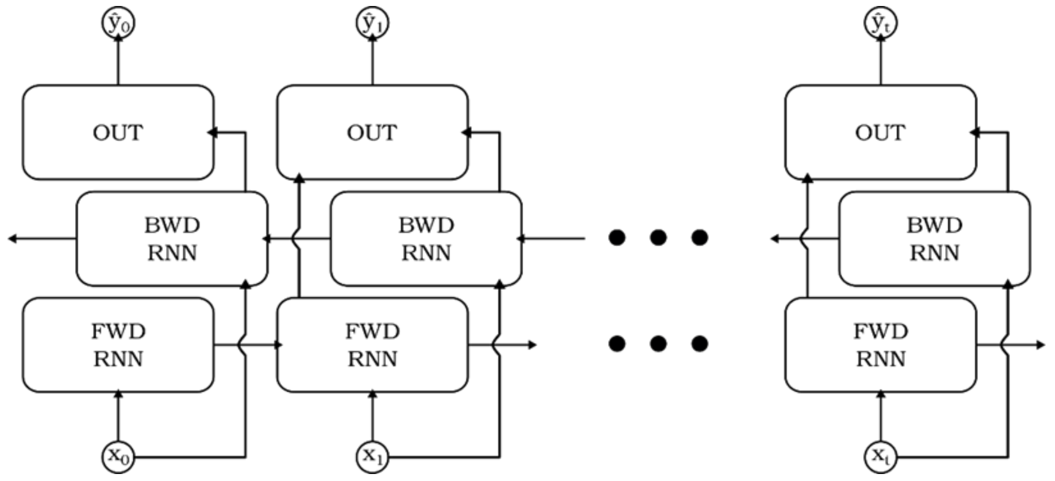


Figure 8 A simplified depiction of a bidirectional RNN. Sequential dependence information is passed on in both directions, forward (FWD) and backward (BWD) and later combined in an output layer (OUT).

Figure 8 shows the simplified depiction of a bidirectional recurrent neural network, as described. Bidirectional architectures are a popular choice for the purpose of ML-SAST, as the literature study in Section 10.5 has revealed. Moreover, some evaluation studies on different deep learning architectures for ML-SAST, seem to support the notion that bidirectional models offer a better predictive performance than their unidirectional counterparts[21]–[23].

2.3.8 Graph Neural Networks

Graph Neural Networks (GNNs) represent trainable, parametric functions over graphs[24]. Conceptually, they are more involved, as they allow better correlations over distant features. As an example, aggregate-combine **GNNs** with L layers are specified with aggregate functions agg_i, combination functions comb_i and a classification function CLS. On an input graph $G = (V, E)$, where V is the set of vertices and E the set of edges connecting the vertices, i.e., tuples over $V \times V$, a GNN maintains a state in form of a k-dimensional vector, $x_v \in \ R^k$ for each vertex v. Computation consists of updating these states throughout L iterations, with x_v^i denoting the states after iteration $i \in N$. The computation model corresponds to updates x_v^i (:=) $\mathrm{comb}_i\ (x_v^{i-1}, \mathrm{agg}_i\ \ x_w^{i-1}$ ({{ $w \in N(v)$}})), where N(v) is the set of neighbors for vertex v in G, and {{… }} denotes a multiset (i.e., unordered set whose elements are associated with multiplicities). That is, at stage i, each vertex v receives the state of its neighbors which are then aggregated, and the result combined with the current state x_v^{i-1} to produce the next state x_v^i. The fact that agg_i maps multisets of states into real vectors means that it does not depend on the source of the received messages. GNNs are used for node or graph classification. In the first case, after the final stage, node v is classified into class $\mathrm{CLS}\left(x_v^{(L)}\right)$ determined by a function CLS: $R^k\ \rightarrow\ \{0,1\}$. In the second case, the CLS function maps the multiset

$\{\{x_v^{(L)} \mid v \in V\}\}$ into a single, scalar output; an operation referred to as a readout. The functions involved in the mapping from inputs to outputs can be linear or non-linear, and they are all trainable: in the supervised case by minimizing an error function defined using a training set.

This is merely a very concise explanation as to how this type of neural network operates. Graph neural networks will be covered in much more detail in the ensuing Chapters 11 and 12.

2.4 Other Concepts and Further Reading

A few highlighted algorithmic concepts to machine learning are worth mentioning and often not well understood, or leading to confusion, as there are metric spaces applied in the algorithms not to be mixed with metrics for the evaluations of tools, which will be discussed later on.

Regularization is a general method to avoid **overfitting** by applying additional constraints to the weight vector. A common approach is to make sure the weights are, on average, small in magnitude: this is referred to as **shrinkage**.

The evaluation of different ML algorithms is highly important. One approach applied to supervised learning is **10-fold cross-validation** with 90% of training, 10% of test, executed 10 times. One other extreme is the Leave-One-Out evaluation.

In unsupervised learning, there is no training data with predefined classifications that can be used for orientation. These can be found, for example, through clustering. In addition to the feature vector, a **distance measure** (a so-called **metric**) always plays a role. Simple metrics for numerical data is the Euclidean distance, derived from the so-called **2-norm**, or another $\boldsymbol{p}$**-norm** (e.g., sum total or "infinity" norm). In the case of strings, the **Levenshtein** or **editing distance** could help, which turns out to be the set of editing operations and can be found with dynamic programming.

Of course, linguistic peculiarities can also be used here (removal of filler words). A frequency analysis in texts called **bag of words** (BOW) can also lead to a measure of distance (often used with spam filters).

For time series, there are similar distance measures as those of the **dynamic time warp** (DTW) procedure, which is based on the approximate character string search. There are similarities between DTW and the **Needleman-Wunsch** algorithm (global alignment) and the **Smith-Waterman** algorithm (local alignment). Distances between more complex data, e.g., in tree structures or graphs, can also be found using feature vectors.

As soon as a metric space (M, d) with **distance measure** d in $\mathbb{R}$ is available, machine learning processes can be used. More precisely: A mapping $d\colon M \times M \rightarrow \mathbb{R}$ is called a metric on $\boldsymbol{M}$ if the following axioms are satisfied for any elements x, y and $\boldsymbol{z}$ of M:

1. $d(x, y) = \mathbf{0}$ if $\boldsymbol{x=y}$
2. $d(x, y) = \mathbf{d}(\mathbf{y}, \mathbf{x})$ (symmetry)
3. $d(x, y) \leq d(x, z) + d(z, y)$ (triangle inequality)

Please note that from the axioms follows $d(x, y) \geq 0$ for any x, y of M (positive definiteness).

The aforementioned definition applies to all the distances considered above. For clustering processes, inter- and intra-cluster distances are measured and linked to a target value that needs to be optimized in order to obtain an at least locally optimal clustering (**k-means**). Also hierarchically organized agglomerative clustering (e.g., dendrograms) or **chain-based clustering methods** are based on a suitably selected distance measure. Later in the course of this study, clustering approaches to ML-SAST are briefly described that use distance metrics (see Section 4.1).

The list of concepts is neither sorted nor complete. The reader is referred to the literature. There are several **textbooks** on machine learning, like Peter Flach. *Algorithms that Make Sense of Data*. Cambridge, 1. Edition (2014) [25], Tom Mitchell: *Machine Learning*, McGraw Hill, (1997) + New Chapters (Web) [26], Pat

Langley: *Elements of Machine Learning.* Morgan Kaufmann (1995) [27], Shai Shalev-Shwartz, Shai Ben-David. *Understanding Machine Learning.* Cambridge (2014) [28], Christopher M. Bishop: *Pattern Recognition and Machine Learning.* Information Science and Statistics, Springer (2006) [29], Richard Sutton, Andrew Barto: *Reinforcement Learning*, MIT (1998) [30], together with recent publications in A/A*-level AI conferences (e.g., AAAI, IJCAI, ECML, ICML).

3 Static Application Security Testing

Static Application Security Testing (SAST) is an approach to security testing of software, in which low-level programming bugs, such as buffer overflows, simple race conditions, code injection vulnerabilities, and cryptographic errors, are detected statically. Chapter 7 describes these various types of errors in detail. This means that security checks are performed directly on the (source) program code, as opposed to a dynamic analysis, which is carried out when the program is running.

In the following, information is provided on the technical background of SAST tools. More details on such techniques, for example, can be found in Graf's dissertation[31] and in Chapter 4 of a book on SAST by Chess and West[2].

3.1 SAST Tools

SAST tools often employ advanced static program analysis and related data structures as their foundation. For example, they may use a **taint analysis**, which tracks the flow from an **entry point (source)** to the **vulnerable code (sink)**.

The static analysis engine, however, is only one part of modern SAST tools. Furthermore, SAST tools often do support more than one programming language, e.g., C/C++, Java, C#, but also (to a certain degree) dynamically typed languages, such as JavaScript, PHP, Python or Perl. In addition, SAST tools should support specific software frameworks, e.g., Spring, Qt, or Android, as these frameworks have incorporated specific programming models, such as the Android component framework. Static analyses should reflect the properties of such software frameworks and their programming models.

Most SAST tools use a rule engine that allows an analyst to define rules for various common classes of security bugs like **buffer overflows**, **heap overflows**, **SQL injections** and **cryptographic weaknesses**. In addition, such tools provide a graphical user interface or an input language (e.g., XML, JSON) to define one's own application-specific rules.

Another feature that SAST tools may support is a visualization of how data flows through the code, e.g., from an entry point to the vulnerable code. An analyst can then follow this path and better understand the actual problem or whether it is a security problem at all.

Furthermore, SAST tools serve a didactic purpose by explaining the current problem (including examples) in detail. So, a developer who is not familiar with a particular class of vulnerabilities is able to study its explanation in detail and compare it with the vulnerable code location.

SAST tools usually allow a security tester to generate a report on the vulnerabilities found in the current project. These reports also contain statistics about the identified (potential) bugs and their severity (e.g., critical, high, medium, and low). Hence, SAST tools conduct an implicit risk estimation and rank the findings along with these categories.

Given the inevitability of false positives and negatives, which are explained in Section 3.2and 8.2, SAST vendors sometimes offer a hybrid approach by combining SAST tools with **dynamic application security testing (DAST)** functionality. Results obtained by the SAST engine can, for example, be confirmed by the DAST functionality.

Many SAST tools are available, often with foci on different aspects, e.g., programming languages, software frameworks, build environments or Cloud-based solutions versus offline tools. A comprehensive overview of the SAST market, including tool descriptions and the tools' capabilities, can be found on the Web site by Gartner[1].

[1] https://www.gartner.com/reviews/market/application-security-testing

There are also open-source tools available, which can be adjusted to one's needs. One open-source tool that supports C is the Clang Static Analyzer, which is based upon the LLVM infrastructure and provides simple static analysis like **null pointer dereference**, insecure usage of C functions (e.g., `strcpy`, `strcat`), and **array-out-of-bounds checking**. Another open-source tool supporting C is the Joern tool[2]. Joern implements static analyses based on a specially-tailored graph data structure and provides some ML functionality.

3.2 Background on Static Program Analysis

In general, static program analyses necessarily produce false positives or false negatives. The theoretical background of this observation lies in Rice's theorem[1]. It states that **proving a non-trivial program property** is, in general, **undecidable**. This can be shown by transferring the problem to the **halting problem**. Therefore, a general algorithm cannot solve seemingly simple problems like null pointer access or array bounds error checking.

False positives are wrongly reported errors, whereas false negatives are actual errors that the static analysis fails to report. The former may overwhelm security analysts who employ static code analysis tools, which may finally lead to a low acceptance of the tool and in the worst case the tool not being used at all. The latter is problematic because the analysis may overlook important errors, giving the reviewer a false sense of security. Consequently, a reasonable trade-off between both error rates is necessary, and today's tools are trying to improve this trade-off. If this trade-off is handled appropriately, static program analysis can be very valuable to improve software quality and security. Approaches like ML-SAST are expected to contribute to a further improvement of the results produced by SAST tools.

3.2.1 Compiler Construction Techniques

SAST builds upon techniques and data structures developed in the fields of compiler construction and program analysis over decades. A compiler usually consists of several phases. First, a program is broken down into tokens, which correlate with words and punctuation in a natural language. This step is called **lexical analysis**. Thereafter, the syntactic structure of the source code is checked, which is called **syntax analysis** and is done by a parser. Typical code structures to be analyzed are if-statements, while-statements or assignments. The third step of the compiler pipeline is the semantic analysis, which, for example, checks whether variables and functions are used with the correct types, so-called **type checking**.

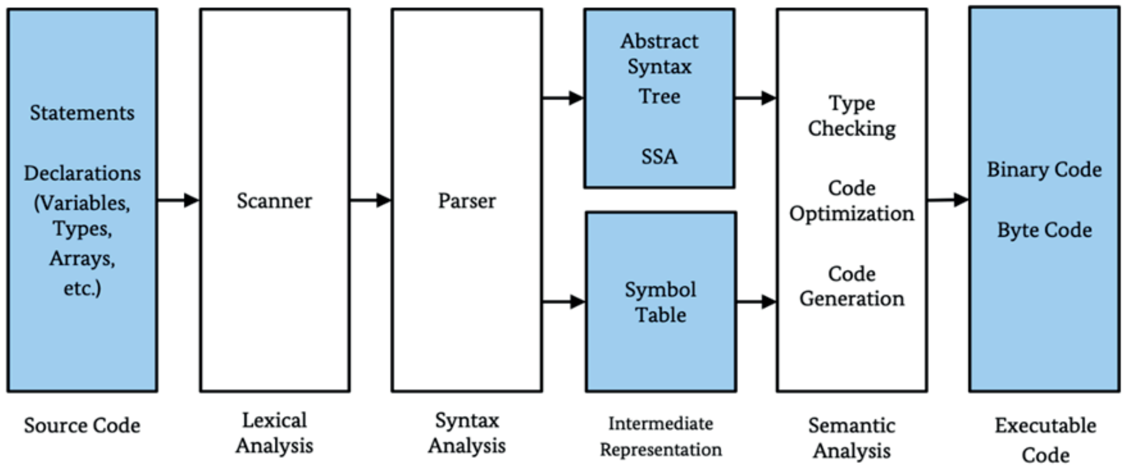


Figure 9 The phases of a compiler.

As a result of the first three steps, an **intermediate representation (IR)** of the program code is built. Typical IRs are **abstract syntax trees (ASTs)**, **stack code** (as in the Java bytecode), and **static-single assignment (SSA)**[32]. SSA is a standard IR that many static analysis frameworks, such as Soot[33],

[2] https://joern.io/

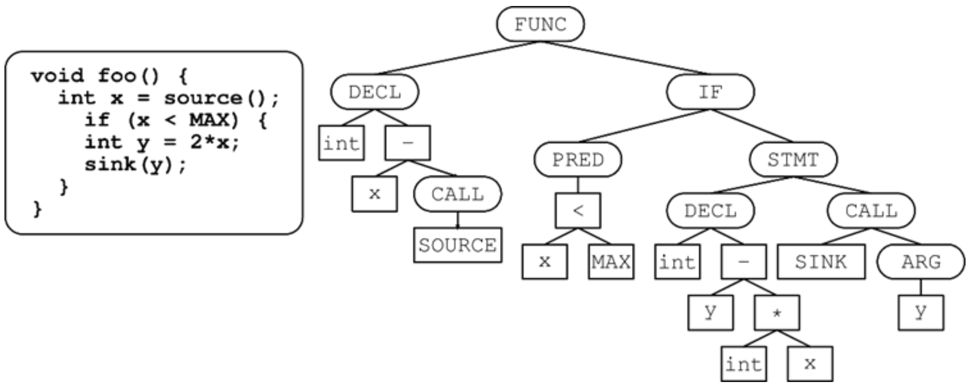
WALA[34], and LLVM[35], support. In addition, due to the simple structure of SSA instructions, the SSA form simplifies control- and dataflow analyses.

Having translated the source code into an IR, code optimizations, e.g., constant propagation and instruction reorder can be conducted, which leads to more efficient binary code. Beyond these code optimizations, many other types of static analyses can be carried out on this IR and related graph structures (see Section 3.2.2 for more information on such graph structures). In the following subsection, some more details on such analyses are given as this lays out the foundations of SAST tools as well.

3.2.2 Typical Internal Data Structures for Static Program Analysis

As indicated above, for static analysis, the source code is often translated into an intermediate representation by means of compiler construction techniques. During the compiler-construction phases or directly from the IR, other data structures can be generated, which may support static code analysis.

One standard data structure are **abstract syntax trees (ASTs)**. In a tree-like fashion, they are usually constructed by a parser during the syntax analysis phase and represent program statements, such as for-loops, if-statements, assignments, and conditions. In this way, ASTs correspond to parse trees but truncate non-informative symbols by means of abstraction, e.g., braces and semicolons. An example of an AST is depicted in Figure 10. On the left-hand side is a short C program, whereas on the right-hand side is the corresponding AST. One can see that syntactic symbols like braces and semicolons are missing; only information important for the semantics of the program are represented.

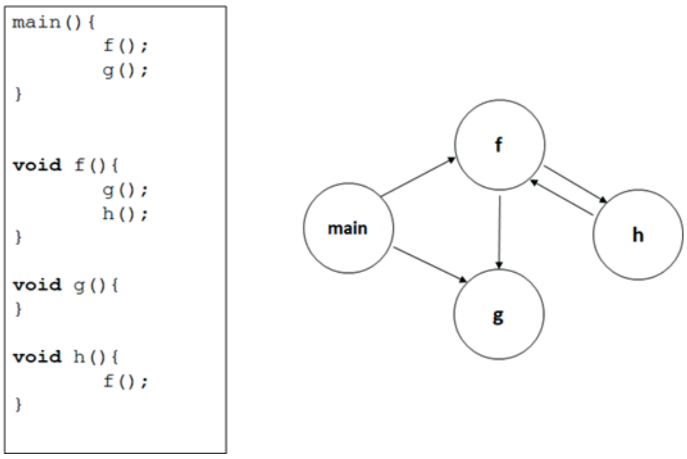


```
void foo() {
  int x = source();
    if (x < MAX) {
    int y = 2*x;
    sink(y);
  }
}
```

Figure 10 On the left-hand side, an example program is shown, and on the right-hand side, the corresponding AST is presented (figure taken from Yamaguchi et al.[36]).

Often **graph-based data structures** are valuable for static analyses. Typical graph structures are **control-flow graphs (CFGs),** representing how the control passes through the statements, e.g., by following conditional and unconditional jumps. CFGs are usually used within a function or method and hence represent intraprocedural control flows. However, inter-procedural variants are possible as well.

Call graphs (CGs) represent the dependences of method/function calls in a program. Depicted on the left-hand side of Figure 11, one can see a simple example program with four functions main, `f`, `g`, and `h`. In particular, `main` calls `f`, `f` calls `g` and `h` calls `f`. The corresponding context-insensitive call graph is shown on the right-hand side, where one can see the call dependencies represented by a directed graph.

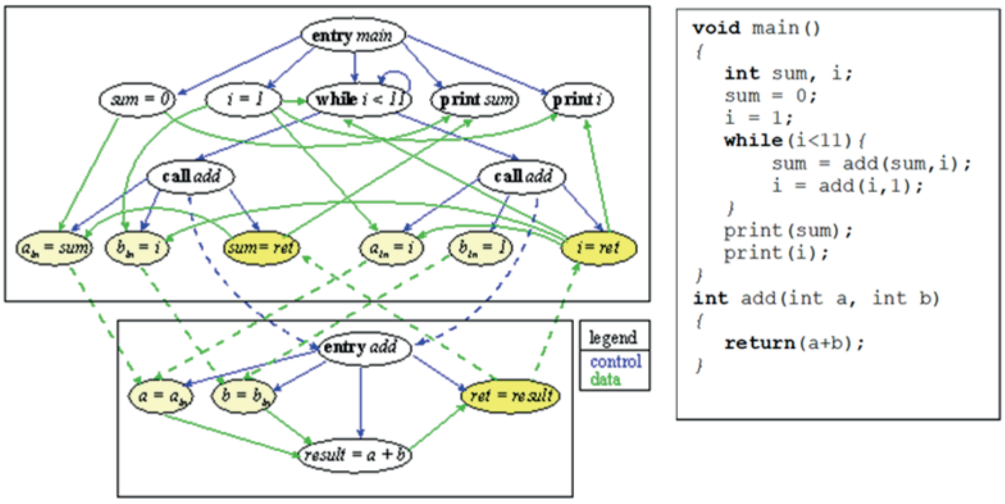


```
main(){
        f();
        g();
}


void f(){
        g();
        h();
}

void g(){
}

void h(){
        f();
}
```

Figure 11 On the left-hand side, an example program is shown, and on the right-hand side, the corresponding context-insensitive call graph.

Often, analyses also need data dependencies, i.e., one calculates which variable definitions flow into which variables, fields, arrays, or parameters. Such information can be represented by a **data flow graph (DFG).**

A program dependence graph (PDG) is of interest if a static analysis needs both: control and data flow. A PDG combines both control flow and data flow information. PDGs only consider intraprocedural control- and data flow. If one needs an inter-procedural variant, one can use a **system dependence graph (SDG)** as having been introduced by Horwitz et al. [37]. Figure 12, which has been taken from [38], depicts an exemplary SDG (on the left-hand side). The corresponding C code is given on the right-hand side of Figure 12. Here, two C functions, namely `main` and `add`, are defined. The function `main` calls `add` with two parameters, whereas `add` returns the result of the addition of its two parameters. The blue edges then represent control flow, and the green edges data flow dependencies.



```
void main()
{
   int sum, i;
   sum = 0;
   i = 1;
   while(i<11){
      sum = add(sum,i);
      i = add(i,1);
   }
   print(sum);
   print(i);
}
int add(int a, int b)
{
   return(a+b);
}
```

Figure 12 An exemplary SDG on the left-hand side and its corresponding C program on the right-hand side (figure taken from [38]).

Sometimes it is useful to combine different graphs for program analysis. Yamaguchi et al., for example, combine ASTs, CFGs, and PDGs into a joint graph structure, called **code property graph (CPG)** and perform their analyses using graph queries [36]. Chapter 4 elaborates on the origins and usage of CPGs in more detail. In addition, the Joern tool, which has been mentioned above, uses the CPG as a key data structure for its analyses.

3.2.3 Static Analysis Problems

Having IR and related data structures at hand, specific static analyses can be defined. Usually, these analyses use **control flow and data flow analyses**[39], i.e., they track control and data flows using the aforementioned graph-based data structures. Usually, one differentiates between two kinds of analyses: **intra- and interprocedural analyses**. The former is carried out within a function, whereas the latter tracks data and control flows along with several functions. Typical examples of static analysis problems include the detection of null pointer violations or array out of bounds accesses.

One important analysis in languages with pointers and references are pointer analyses, which determine whether two or more pointers refer to the same location (e.g., on the heap). Therefore, this analysis is also often called “alias analysis”. Listing 1 shows a small C program where two pointers `p` and `q` point to the same location. If one, for instance, changes `q->x`, then `p->x` is modified accordingly. Given that a precise analysis is not possible, certain simplifications are necessary, accepting the fact that false positives may be produced[31].

```c
#include <stdlib.h>

struct point { int x, y; };

void f() {
    struct point *p, *q;
    p = (struct point*) malloc(sizeof(struct point));
    p->x = 5;
    p = q;
    q->x = 42;
    /* => p->x == 42 */
}
```

Listing 1 The variables `p` *and* `q` *are aliases, and hence changing* `q->x` *also changes* `p->x`*.*

Another application of data and control flow analysis is **forward and backward slicing** [40]. Starting from a seed statement s, forward slicing calculates those statements that s influences transitively. Backward slicing then transitively determines all statements that influence a given seed statement s. Slicing algorithms employ SDGs, which contain inter-procedural data and control flows, as proposed by Horwitz et al. [37]. Formally, Horwitz et al. define (**backward) program slices** as follows:

> *“A slice of a program with respect to a program point p and variable x consists of all statements of the program that might affect the value of x at point p.”[41, p. 1]*

Certainly, static analyses can be devised for specific problems, such as null pointer violations, array bounds or use after free errors. Again, one can differentiate between inter- and intra-procedural versions of the algorithms, depending on the concrete problem to be solved.

4 ML in the Context of SAST

The application of machine learning for SAST is an emerging field in the discipline of computer science. Its roots may be traced back to several key developments, beginning in 2014. This chapter elaborates on some of these developments in an attempt to draw a comprehensive timeline. Specific ML-SAST approaches will be picked up again in later sections and explained in-depth.

4.1 Preliminaries

Research concerning the application of machine learning for SAST can be traced all the way back to two publications by Yamaguchi et al. in 2014 and 2015. The first publication discusses a new composite code representation that combines abstract syntax trees, **CFGs** and **PDGs** into a single heterogeneous structure. These so-called **code property graphs (CPGs)** can efficiently encapsulate code semantics and present a well-established building block in the context of ML-SAST[36].

Yamaguchi et al. themselves went forth in 2015 and demonstrated the capabilities of their CPGs in a follow-up paper. Here the graphs are loaded into a graph database where an algorithm would then extract paths indicative of a vulnerability[42]. This algorithm rests on the principle of **program slicing**, as introduced in Section 3.2.3. Informally speaking, these slices present a condensed form of the code that has been stripped from any semantically superfluous statements. It might be for this very reason that program slicing has been widely adopted in the field of ML-SAST. It allows for the efficient extraction of relevant features from the code, suitable for model training. In particular, this is important as superfluous data might lead the model to infer false causalities from these artefacts[43, p. 3].

Another relevant publication that gained notoriety was published in 2016 by Wang et al., who may have been the first to leverage **deep representation learning**, i.e., the employment of deep learning methods for the automatic inference of relevant features, in this case, the important semantic constructs from source code. The authors use tokens extracted from abstract syntax trees to learn a deep belief network to predict code defects. Deep belief networks are a type of neural network with several highly connected layers. According to their own evaluation, the approach showed promise and thus seems to have paved the way for subsequent research efforts[44].

These first attempts have laid the foundation for some important publications in the field of ML-SAST. In summary, their contributions were: 1) the proposal of CPGs, a code representation structure capable of precisely capturing a program's semantics; 2) the use of program slicing, a procedure for efficiently dissecting source code into semantically coherent segments and 3) showing the potential of deep representation-learning for code analysis.

4.2 Refinements

One of the most influential works to come out of this general ML-SAST approach is a paper by Li et al., titled "VulDeePecker". The publication appears to be one of the earlier examples that specifically targets the detection of vulnerabilities in source code using deep neural networks. For this, the authors employ **bidirectional long-short-term-memory networks (BLSTM)** as the core technology. BLSTM, combined with the application of so-called code gadgets, a concept similar to that of code slices, may have led to the popularity of this publication[45].

The publication of the VulDeePecker paper has caused a high number of derivative works to be released that either partially incorporates the methodology employed by Li et al. or use the paper's results as a baseline to measure the improvements of their own approaches[46]–[51]. In addition, the authors of the VulDeePecker-paper themselves went on to publish follow-up papers, investigating different facets of their approach to improve on.

The first paper to be released was named "SySeVR" and proposed a new and improved form of code gadgets, called **syntactic vulnerability candidates (SyVC)** and **semantic vulnerability candidates**

(SeVC). SyVCs reflect the syntactic aspects indicative of a vulnerability, e.g., the character * denoting the declaration of a pointer, and can be inferred through pattern matching. SeVCs, on the other hand, are generated based on the SyVCs by means of program slicing on the **Program Dependence Graph (PDG)**[52]. These ideas were later refined in another paper by the same authors, called “VulDeeLocator”. Here the authors enhanced the code representation of syntactic and semantic vulnerability candidates, using an intermediate code format and narrowed down the detection granularity to three lines of code[45, p. 2]. Yet another follow-up publication by Li et al. aimed at a **multinomial classification** approach that not only allows for the detection of vulnerabilities but is also capable of inferring the kind of vulnerability present in the code[53]. Unlike the previously mentioned ideas, it appears as if multinominal classification for the detection of vulnerabilities was not pursued any further after that.

4.3 Current State of the Art

These publications have kickstarted the research on ML-SAST. Many works following the advent of the VulDeePecker paper, however, seem to closely adhere to its established pipeline. There are several problems to this approach, however, that, unfortunately, have been disregarded for the most part. At a closer inspection, these problems come down to the following three key aspects.

4.3.1 Datasets

Unfortunately, up until recently, little attention has been paid to the datasets the models are being trained on. As several evaluation studies on popular approaches, such as VulDeePecker, have demonstrated, synthetic datasets appear to be unfit for the task. When the authors of these evaluation studies tested the different approaches against real-world data, their performance dropped by a large factor across all metrics[43, pp. 8–9], [54, p. 9]. More importantly, preprocessing steps, such as code normalization, that is, the removal of individual identifiers from the code, as well as the procedures employed for the code slicing, may have led to data duplication in the datasets[54].

4.3.2 Architectures

With respect to the neural network architectures utilized in the literature, recurrent type neural networks appear to be the ones most favored. Here, especially **bidirectional LSTM networks** seem to dominate the landscape. This may not come as much of a surprise, as the contextual awareness of such networks makes them perfectly suited for processing sequential data, which source code may be attributed to as well. Nevertheless, there are also problems associated with their use. For instance, the features extracted from the source code must be embedded into a shallow data structure for the LSTM networks to be processed. These data structures, however, fail to preserve important semantics [55, p. 3]. For this very reason, other authors suggest the usage of an entirely different type of neural network, called **graph neural network (GNN)**. When training such GNNs, no explicit embedding of the graph structures is needed. Instead, the graphs can be fed directly into the model, allowing for a highly efficient mode of representation [56, p. 2].

4.3.3 Explainability

Lastly, another challenge concerning the generalizability of these models appears to arise from the difficulty of explaining their inner workings. In the case of the VulDeePecker, the model seems to have picked up on artifacts during training, causing the predictions to be made based on false causalities[43, p. 9]. Notwithstanding, the latest developments in ML-SAST seem to indicate that these problems finally come to the fore, as Warnecke et al., for instance, conducted a comprehensive survey carrying the title “Evaluating Explanation Methods for Deep Learning in Security”. Here, the authors explore the applicability of methods previously used for the explanation of deep learning approaches for image processing in the context of information security[57].

4.3.4 Outlook

As outlined, developments in the research area of ML-SAST have been rapidly evolving. When Yamaguchi et al. introduced their novel **code property graphs** and later a method that automatically infers vulnerability patterns from these, they laid the foundation for ML-SAST[36], [42]. Building upon this, first Wang et al., closely followed by Li et al., were able to show the potential of utilizing **representation learning** for the detection of software defects and vulnerabilities in source code. As of late new evaluation studies by Arp et al. and Chakraborty et al. went on to draw a first résumé, summing up the current state of ML-SAST and pointing towards severe shortcomings concerning those previous approaches[36], [54]. As it appears, there are three major areas, requiring further research. Firstly, there is a need for more adequate datasets that the models can be trained on. Alternatively, clustering approaches, such as that of Yamaguchi et al., may circumvent the need for such labeled datasets. Secondly, better embedding procedures capable of encapsulating the code's semantics and models need to be developed. In this regard, the works by Cheng et al. and Zhou et al. that relied on the use of graph neural networks, may show some promise[55], [56]. Lastly, methods may be explored that aim at making these models behavior explainable. Warnecke et al. were the first in that regard to successfully apply such methods in the area of ML-SAST, demonstrating the feasibility of such endeavors[57].

5 Expert Interviews

To obtain a better overview of how SAST tools are used within the process of security assessment of C/C++ software in practice, interviews with several experts from industry, government agencies as well as academia were conducted. The focus of these interviews was to gain insights into the benefits as well as the limitations of current SAST technology for C/C++ in daily usage of security testing. Within the frameworks of these expert interviews, typical **error use cases** that occur in C/C++ code which are hard to detect by current commercial SAST tools were queried. The interviews concluded with a discussion of possible ML-based solutions to these problems.

In the following, the procedure of conducting the interviews is described. This way, the reader gets an impression of the interviews' process. Secondly some information on the interview partners (e.g., concerning job functions, experience, kind of organization they are working for) are presented. The section concludes with the results of these interviews, including some conclusions drawn from them.

5.1 Method

The interview method is partly based on the procedures developed by Gläser and Laudel[58]. This method was applied within the frameworks of another study before, which dealt with the relevance and distribution of security patterns in practice[59]. In particular, the concept of anchor questions (leading questions) was adopted.

As a preparation for the interviews, a guideline that covered the following five main topics was designed:

1. The background of the interviewee concerning the programming languages C/C++,
2. the used C/C++ versions (e.g., C 99, C 11, or C 17),
3. SAST tool usage and preferences,
4. the strengths and weaknesses of SAST tools in general, and
5. the expected functionality of (future) ML-SAST tools.

In total, 18 questions were defined for the interview, distributed over the aforementioned five thematic areas. The appendix contains the complete translated guideline since the original version was written in German. Additionally, the following four **anchor questions** were defined:

1. Which C versions currently have a high distribution among developers?
2. Which SAST tools are well-suited for the analysis of C code?
3. What requirements are important to you that a ML-based SAST tool should meet (e.g., low false positive / negative rate, high performance, explainability)?
4. Can you give typical cases that are usually hard to find for conventional SAST tools but are at the same time important for your work (if possible, please give cases that are as concrete as possible)?

The purpose of these anchor questions is to guide the conversation back to the initial questions if the interview should go in the wrong or an unforeseen direction. These questions as well as the complete guideline were sent to the interview partners in advance to give them a feel of what type of information is expected from them. The goal of this step is to make the interviewees feel more comfortable concerning the expected information and to give them the opportunity to prepare themselves for the interview.

Based on the guideline and the anchor questions, each interview was conducted as a video conference. The Jitsi[3] conference system was chosen because it is open source and served the purposes well. A teleconference system was not used as it was important to obtain direct feedback through facial

[3]https://meet.jit.si

expressions and provide a familiar environment for the interview partners – a videoconference seemed to be a better option to use here. For data protection reasons, the videoconference was not recorded; otherwise, some interview partners would have had to request permission from their organizations. Moreover, it would then have a required higher data protection standard to store the recordings of the interviews. Also, the privacy of the interview partners would have been unnecessarily affected. To overcome these limitations, the answers to each question were merely protocolled. A data protection declaration was prepared, which was designed with the help of a company data protection officer. This declaration was to be confirmed by each interviewee. Each interview was carried out with four members of the project team, including the interviewer and a minute keeper. It was also made sure that the interview team covered enough expertise regarding the topics SAST, ML, and C/C++. The interview was designed to last about an hour. The actual interviews showed that this time estimation was realistic.

5.2 Interview Partners

Eight interview partners from different organizations and companies were recruited for the expert interviews. In the following, some anonymized information on the interviewees were given to allow a rough classification of the interview partner's background and experience:

Interview partner #1 is responsible for software security in a small or middle-sized enterprise (SME). This SME develops security-critical code, with a focus on e-government software, such as eIDs.

Interview partner #2 works for an IT service provider for the military. He has a strong background in ML and experience in C/C++.

Interview partner #3 also works for the IT service provider for the military. He has experience in C/C++. In addition, he has a strong background in reverse engineering and exploit development.

Interview partner #4 graduated with a PhD in the field of software security and machine learning. Currently, he is working for a static program analysis company.

Interview partner #5 works for a German authority on information security. He has much experience with the security of C/C++ code as well as malware detection.

Interview partner #6 is now a full professor for Cybersecurity. Prior to that position, he was responsible for introducing SAST tools for a large international software vendor.

Interview partner #7 works for a significant security consulting company. He has experience in static program analysis with a focus on formal approaches as well as on ML.

Interview partner #8 answered the guideline digitally, due to time constraints. He is a full professor for information security, with a very strong background in employing ML in the field of information security for more than 15 years. His answers are incorporated as good as possible in this section. However, some questions could not be asked in the survey and thus are missing.

5.3 Results of the Expert Interviews

First, participants were asked to state the earliest C/C++ versions that should be supported by a possible SAST tool with machine learning support. The interview partner could also report the version that is actively used at the corresponding company or institute. The interview partners responded with the following answers:

5.3.1 Versions of C and C++ Used

Table 1 Versions of the C and C++ programming languages used by the participants of the interview.

Participant	***Comment on the C and C++ versions used***
1	The participant reported that the current C++ standard is used at the company. At the time of the interview, this was C++ 17. The participant did not report on a specific version the tool should support.
2	All current C versions are used at the company the participant works for.
3	The participant reported version C 99. He further elicited that the developed tool should run on most Linux systems.
4	Older versions of the programming languages should be supported, as some security problems have already persisted for some time.
5	The newest standard of the programming language should be supported (C14). From the C99 version onwards, the programming language did not change significantly anymore. The participant believes that the version is not so critical, as they are backwards compatible.
6	The interview partner considers C90 to be the standard version. The participant suggested that older and classical versions should be supported by a SAST tool.
7	The participant stated that at least C99 should be supported.
8	The participant stated to only use C99.

Out of the eight participants one responded on the C++ versions and seven on the C versions. Concerning C++, the participant noted that the most current standard of C++ is used at her organization. Regarding a specific version, which should be supported by a possible ML-SAST tool, no specific version was mentioned. The responses on the C version showed some similarities. The participants reported that older C versions should be supported. One participant justified this statement with the fact that security issues have persisted for some time.

5.3.2 Opinions Towards SAST Tools in General

Table 2 Opinions expressed by the interview partners concerning SAST tools in general.

Participant	***General opinions expressed on SAST Tools***
1	SAST tools have good code smell detection capabilities and give good hints, but they also report many false positive errors.
2	As C is often used in critical infrastructure projects, SAST tools should be used to exclude security problems. The participant further pointed out that SAST tools might have difficulties with programming projects that have many dependencies. The results of SAST tools should be reasonable. The usage should be highly automated, have a high quality and high efficiency.
3	SAST tools can support when dealing with high-level programming languages – which have a long call graph
4	The users of SAST tools can only benefit from such tools if they understand the reasoning behind the detected problems. The users should not just use the tool but also be able to define rules for the tool to detect problems in the given software context.
5	SAST tools can help to enforce rules and coding standards. Such tools also produce many false positive errors.

Participant	***General opinions expressed on SAST Tools***
6	SAST tools can be integrated at an early stage in the development process. They help to detect errors in the source code without compiling or running the program. Not much expertise is needed to use such tools. To use such tools, the source code needs to be parsed. It is therefore of great interest which C standards should be supported. Expert knowledge is necessary to evaluate and classify errors as false positives. As these tools are frameworks, they need to be configured for personal or team usage and should not be used as an off the shelf product. As these tools do not support all programming paradigms, they might produce false positives or false negatives.
7	SAST tools can help to verify program properties, exclude runtime errors, and many other errors. The proof of program properties is difficult. Under- or over-approximation might lead to many false positives and false negatives. Configuring such tools is difficult. The multiplication of two variables is not decidable.
8	N/A

Many of the experts noted that SAST tools show a high number of false-positive errors. This corresponded with findings from literature. Some false positives can be explained with the fact that static analysis techniques need to make some approximations and simplifications (see Section 3.2.3). The usage of such tools might be easy, as they can be easily integrated into a development process, but to interpret the results and classify errors as false-positives expert knowledge is needed. To complement this statement some experts noted that these tools should be used as a framework. They should not be used as an off the shelf product but should be configured depending on the project.

5.3.3 Input Format for Rules for SAST tools

Concerning the format of rules for SAST tools, a few responses were given. One participant pointed out that rules for the tool Fortify are defined as finite state machines. As these rules represent the basis of business they are encrypted. The same participant mentioned that rules for the tool Checkmarx are defined as C# source code, which makes them easy to be understood and extended by software developers. A second participant mentioned that rules can also be defined in the form of a query language. The tool joern.io uses rules that are defined in a Scala-based query language.

5.3.4 Expectations Towards ML-based SAST Tools and Ideas to this End

In the following table, expectations and ideas that the interviewees formulated with respect to future ML-SAST tools are presented:

Table 3 General ideas and expectations towards ML-based SAST approaches, shared by the participants of the expert interviews.

Participant	***Expectations and ideas concerning ML-SAST tools***
1	Reduced false positive rates, explanation for detected errors should be presented.
2	The participant proposed a combination of supervised and unsupervised learning algorithms to train a machine learning model. First, a clustering algorithm should be used to group the data. A supervised learner is trained using these classes. The participant further suggested that using a reinforcement learner might further improve the classification performance.
3	The participant suggested learning rules from the source code. Unsupervised learning, graph and tree-based approaches might be useful. The participant believes reinforcement learning might be purposeful.

Participant	***Expectations and ideas concerning ML-SAST tools***
4	The participant suggested that collecting enough training data might be difficult. Labelling the data might be even more difficult. Furthermore, the training data might depend on the context. For example, should there be training data per CVE error class or training data for each instance of the CVE error class? For supervised learning, the expenditure to collect data might even be higher. For example, to train a supervised learning algorithm, many labelled training cases are needed. Contrary to this, it might be easier to write a simple rule to detect the problem.
5	The participant noted that explainability is very important. The user should be informed how the tool decided to rate a certain part of the source code as not safe. Furthermore, the participant noted that selecting the right features for a machine learning algorithm is very important. It could either be trained on tokens or an abstract syntax tree of the source code. Unsupervised learning might be a possible approach. The participant mentioned n-gram models, support-vector machines and random forests to be valuable and possible approaches for a machine learning supported SAST tool.
6	The participant mentioned that many SAST tools already claim to have some sort of machine learning functionality. The participant pointed to one SAST tool provider that claims to have such functionality. Though the participant also noted that verifying such a claim, especially for a SaS provider, is rather difficult. The participant continued with the following: “Why learn, if you can do it deterministic”. Nevertheless, he suggested that machine learning might help in the following areas: Analysis of small code patches Analysis of source code with missing context Faster analysis Learning project-specific patterns Detect good Audits The participant noted that a machine learning SAST tool might learn new vulnerability fixes from GitHub and present these to the user. To do so the tool would need to evaluate the commit messages.
7	The participant made it clear that if the tool learned rules to distinguish good from bad code, humans should understand these rules. Furthermore, such a tool should have a low false positive rate. As failure analysis is time-consuming, the number of findings should not be too high. The tool should also be efficient and fast. He has a rather critical view of machine learning and deep learning because their behavior is hidden in a black box. Therefore, the generalization of the results might be difficult. Nevertheless, the participant sees great potential in machine learning, especially regarding natural language processing (NLP). A question of great concern is how to deal with the 1% of errors that are not correctly detected? Success in academia might be easier to achieve than in the industry. The participant suspects good results in inductive logic programming and probabilistic logic programming. Machine learning might be a good complement but is rather not practical and rather used in academia. The participant further mentioned investigating MLP from Google, Competition of Software Verification (https://sv-comp.sosy-lab.org/2021/) and SMT solver (Satisfiability modulo theories).
8	N/A

Some participants of the interview reported that a SAST tool with ML support should have a reduced false-positive rate compared to current SAST tools. Furthermore, the explainability of the detected error was of great concern to some of the interview partners. Some participants mentioned that a ML-SAST tool should run faster than a normal SAST tool. One of the experts noted that collecting enough training data as well as labelling it for a SAST tool with ML support might be difficult. The expert also stated that it might be easier to write a rule to detect an error in contrast to collecting enough training data to learn the error. This statement was supported by another interview partner, who stated: *„Why learn, if you can do it deterministically."*

5.3.5 Easily and Not Easily Detectable Errors

In the following the interviewees' comments with respect to common programming errors and the effort to detect them are presented:

Table 4 Types of program errors easily (a) and not easily (b) detectable by means of SAST, as reported by the interview partners. Some of the participants did not report on any error use cases.

Participant	***Easy to detect errors by SAST tools***	***Hard to detect errors by SAST tools***
1	N/A	N/A
2	N/A	N/A
3	• Buffer overflows • Memory leakage • Bad usage of pointers • "simple cases"	• Use after Free • Eternal Blue (no call-trace for both) • Format strings
4	N/A	• Dataflow in arrays and structs • Dynamic loading of code (missing at checktime) • Source of user input is hard to detect • Dynamic programming languages o Creating a call graph is difficult
5	N/A	• void pointers and object casting in C • Concurrency • Rules work on functions (intraprocedural) but not across multiple functions (interprocedural)
6	N/A	• Pointer arithmetics in C o Expensive tools perform better than cheap ones • Numeric over- or underflows in C • High usage of frameworks can hinder SAST tools
7	N/A	• Interesting program parts will be difficult for the SAST tool o There is always a deviation of a problem a SAST tool will not find • Client – server communication is difficult for a SAST tool
8	N/A	• Static analysis techniques have problems detecting errors that arise from runtime behavior o Some of these are the following "use after free", "deadlocks", "TOCTOU"

Only one participant reported on the question concerning easy to detect errors. The participant mentioned the following cases: buffer overflows, memory leakage, bad usage of pointers and "simple cases". On the other hand, most of the participants were able to present difficult to detect errors. Their opinion towards difficult to detect cases overlapped in the following points. Errors that arise from runtime behavior like "use after free", "deadlock", "TOCTOU", "Eternal Blue", and "concurrency" were mentioned to be hard to detect by static analysis techniques. Furthermore, errors that arise through the interplay of different functions are difficult to detect. To detect such problems an interprocedural analyses needs to be performed. One participant also noted that errors which arise in dynamic programming languages are difficult to detect. According to the expert these errors are difficult to detect because it is difficult to create a call graph for a dynamic programming language. One participant mentioned that errors which arise through void pointers and object casting in C are difficult to detect. The error use case void pointer will be explained in more detail in Section 7.1.

5.3.6 Sources for Error Use Cases

Table 5 Possible sources for error use cases, reported by the interview partners. Some of the participants did not answer this question.

Participant	***Reported sources for error use cases***
1	• Search public repositories for Commits that fix CWEs
2	• Web scrapping for CVEs ○ Find source code
3	N/A
4	N/A
5	• International Obfuscated C Code Contest • The Underhanded C Contest
6	• Obtaining enough data might be challenging • Juliet Test Set might be too specific • NIST has two benchmark sets available • Extract source code from public repositories ○ Search for CVE entries and their fix • Generate test data based on error paradigms ○ Testcases are application specific
7	• Scientific papers • CVE • IEEE • Scientific conferences ○ Principles of program analysis ○ Computer-Aided Verification (CAV) ○ Association for Computing Machinery (ACM)
8	N/A

Several participants noted that web scrapping public repositories for commits, which fix CWEs and CVEs, might be a suitable way to obtain error use cases. One participant mentioned that the National Institute of Standards and Technology (NIST) provides some test sets. Though the participant also mentioned that the Juliet Test Set – which is provided by the NIST – might be too specific to be used as training data. Instead of using these test sets the participant recommended to generate test cases based on error paradigms. Furthermore, scientific papers and conferences, C coding contests as well as the IEEE were mentioned to be used as sources for error use cases. Regarding the question in which format error use cases should be documented and persisted two participants mentioned the typical human and machine-readable formats: yaml, json and xml.

6 Online Questionnaire

Besides the experts' opinions in Section 5, the view of the development community towards the topic was also retrieved. As they use static application security testing (SAST) tools in their daily work, their perspective on the topic is also of great concern. Querying their view was done in hopes to retrieve further error use cases, provide a good overview of the opinions of the development community towards SAST tools in general and as a way to let the community participate in the development requirements of a SAST tool with machine-learning support. Therefore, an online survey was designed and conducted to collect their opinions and possible requirements for a SAST tool with machine-learning support. As the survey was intended to extend the results from the expert interviews it was based on the questions which were defined in Section 5.1. All the questions can be found in the appendix.

6.1 Platform

The online survey was hosted on the platform SoSci Survey[4]. This platform was chosen as it is compliant with the **general data protection regulation (GDPR)** and was also approved by a company data protection officer. With the help of a data protection offer, a data protection declaration in the English and German language was prepared. To participate in the survey, the participants needed to approve the data protection declaration. Otherwise, they could not participate.

6.2 Participants

The survey was distributed among several European companies and a research institute. The candidates were selected based on their focus of work which deals with C and C++ software development. The different targets could be grouped into the following categories:

- automotive industry
- electrical industry
- software development
- security analysis

The contacts to these companies as well as research institutes were provided by the authors of this study and through the BSI. Besides addressing possible candidates directly, other people were also able to participate in the survey. It was openly accessible, and there were no restrictions imposed on the participants.

6.3 Questionnaire

The questionnaire was based on four anchor questions and the 18 questions defined in Section 5. It was designed to be completable within half an hour but could be paused and finished at a later point in time. The participants were able to complete the survey either in German or in English. The participants were asked to answer the questions as closely as possible. The questions were either yes-no, scale, selection, or free questions. At the end of the survey, the participants were also asked if they would be available for a personal interview. The complete survey can be found in the appendix. In the following, the four main categories of the survey are described.

General Questions: The questionnaire starts with a part requesting a self-assessment of their experience in C/C++ software development and their experience with security in the context of C/C++ development. Furthermore, the occupational field and the used C/C++ versions were queried. The personal usage, as well as the usage preference of other teams was queried.

[4]https://www.soscisurvey.de/

SAST tools: In the following section, the participants were asked for their usage of SAST tools for C/C++ as well as for other programming languages. The participants were also asked to specify useful SAST tools for C/C++ using a selection list and a free text input. The section then completes with questions regarding the advantages and problems of SAST tools. Specifically with questions asking for error use cases that SAST tools can easily detect and error use cases that a SAST cannot easily detect.

SAST tools and ML: This section starts with asking for their experience regarding machine learning and the corresponding subject areas. This is followed by a personal assessment of the value machine learning might bring to static code analysis and then asks for specific use-cases of machine learning in static code analysis. The participants were also asked if they knew of any SAST tool with machine learning functionality.

Feedback: The questionnaire concluded with a text field for further comments concerning the topic. Furthermore, the participants could also express their interest in further contributing to the study in an “expert interview”.

6.4 Results

Due to the low completion rate of the survey, the results and statements, presented in the following, should be taken with caution. Nevertheless, they are presented to illustrate parts of the current opinion and some interesting statements from the development community. The focus is on consistency and inconsistency of the statements within the survey as well as between the survey and the expert interviews.

The survey was available from March 2021 until June 2021. There were 527 visits to the survey (including accidental double-clicks, as well as clicks from search engines). 59 participants completed the captcha, but only 26 participants agreed to the data protection policy. Therefore, only 19 participants completed the survey. The results described in the following are based on the participants that completed the survey.

Since it was not possible to retrieve enough responses, some questions were left out from the analysis. These questions queried the personal experience of the participants in the programming language C and their experience concerning security of C code. A question asking for the field of application of the programming language C was also left out. These questions were formulated to categorize the answers according to the reported categories and detect differences within these groups.

6.4.1 C/C++ Versions

In the beginning of the survey the participants were asked to rate their experience in C/C++ and their experience with security regarding C/C++. Following this self-assessment, the participants were asked to rate the usage of different C and C++ versions personally as well as for collaborating teams. The participants were able to rate their usage preference for multiple versions as well as for both programming languages. Regarding the C version, which is used personally, the participants mostly selected C99, followed by C11, while they did not report on C17 at all. For the C version used by other teams, the picture is similar: C99 was selected the most. The second most-often selected C version was C11, while C17 was only mentioned twice to be used by other collaborating teams, see Figure 13.

A clear picture of the most used C++ version could not be drawn from the answers to the survey. Figure 14 shows the usage preference for C++ versions. Older, as well as newer C++ versions, were used in personal and in collaborating projects.

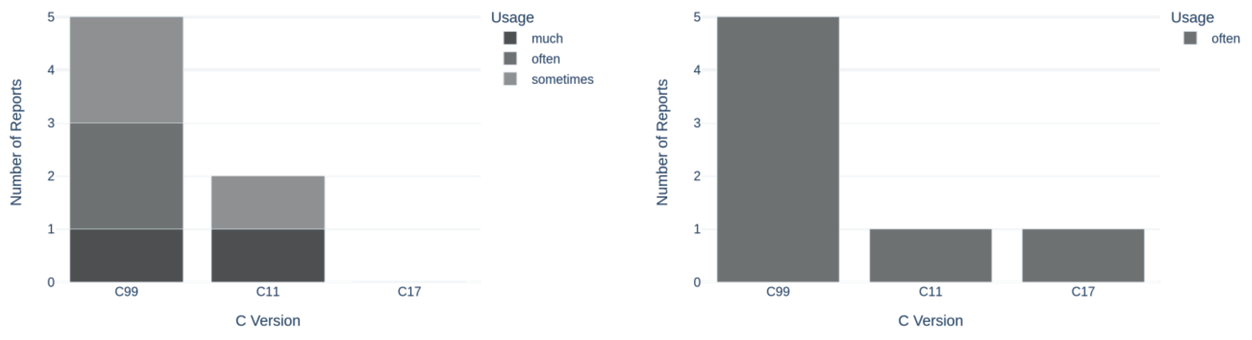


Figure 13 Usage preference of the different C versions. The used C version in personal projects is shown on the left side, while the right side shows the used C version in collaborating teams.

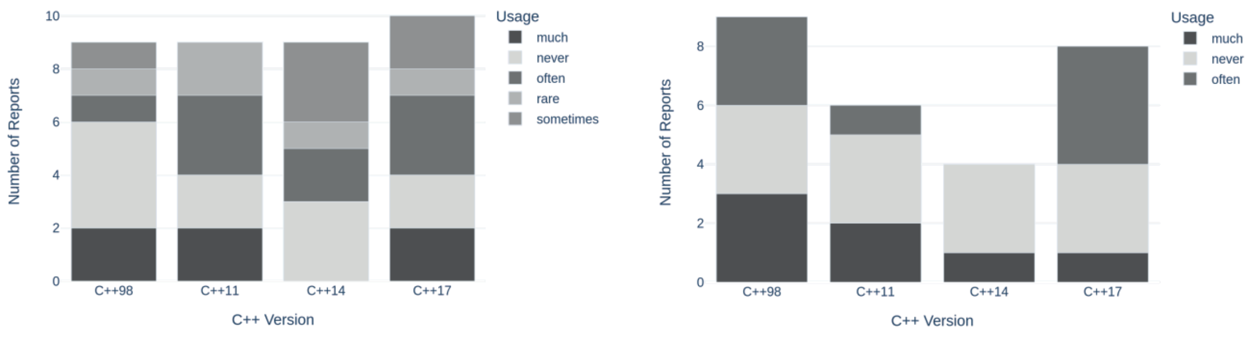


Figure 14 Usage preference of the different C++ versions. The used C++ version in personal projects is shown on the left side, while the right side shows the used C++ version in collaborating teams.

6.4.2 SAST Tools

When the participants were asked to report the advantages and disadvantages of SAST tools, they responded with the following statements.

Concerning the advantages of SAST tools, the participants listed the following points.

- They can scan large codebases systematically.
- A SAST tool makes the source code more readable and maintainable. It also prevents the usage of complex language constructs.
- SAST tools can be integrated into the development process.
- They provide valuable hints for the user.
- Regarding the disadvantages of SAST tools, the participants mentioned the following points.
- As SAST Tools use approximation, they produce many false positives.
- Expert knowledge is needed to classify the reported errors.
- If the users of SAST tools have too little time and knowledge they might discard detected errors and warnings. This comfort/ease might lead the users to even further discard detected errors and warnings and thus the use of the tool becomes pointless.

These points support the findings from the expert interviews. Furthermore, one participant stated that such tools are not always compatible with the newest IDE or C/C++ version.

6.4.3 Input Format for Rules for SAST tools

The participants responded to the question about the format for rules of SAST tools with similar examples as the participants of the expert interviews. Human and machine-readable formats like xml and json as well as plain source code were mentioned. Two examples for the use of source code as rules were also given. C++ source code can be used to define rules for the tool PreFast. Another participant also mentioned that rules can be defined in the form of C# source code for a not named tool. Similarly, to the formulation of rules in the form of a query language for the tool joern.io – mentioned by one participant of the expert interviews – this was also mentioned to be the case for the tool CodeQL. Concerning the input options of the rules the participants mentioned to use command line arguments and the user interface of such tools.

6.4.4 Error Use Cases

The participants were asked to illustrate easy and challenging to detect error use cases. The answers showed similarities and dissimilarities within the survey as well as to the statements from the expert interviews. First, similar statements are illustrated. Buffer overflows were mentioned by two participants of the survey as well as one interview partner from the expert interviews to be easily detectable. A buffer overflow is an error in which data is written outside the boundary of a buffer. The error use case void pointer was mentioned by one survey participant as well as one participant of the expert interviews to be difficult to detect by a SAST tool. A void pointer is a pointer with an undetermined data type. This pointer needs to be cast to its respective data type. More information on void pointers and buffer overflows can be found in Chapter 7.

Apart from these similar opinions, there were also two notable dissimilarities in the statements. First, two survey participants reported that use after free errors should be easily detectable, while a participant from the expert interviews classified this error type as difficult to detect for a SAST tool. Use after free errors arise from dangling pointers. A dangling pointer is a pointer pointing to a memory location that has been released of its original data. If new data is now stored at this memory location and used through the pointer, the underlying program receives invalid data, and thus unwanted results might be produced. For a more detailed explanation, see Chapter 7.

Second, another dissimilarity that emerged was the error use case TOCTOU (time of check time of use). A time of check time of use error arises in the following way. A program first checks if a user has access to write to a certain file. After the check, the program opens the file and writes to it. An attacker might interfere by making a symbolic link from a user-controlled file to '/etc/passwd'. If this is precisely timed, the attacker might be able to overwrite the system password database and thus provoke a privilege escalation. One survey participant classified this to be easily detectable while one participant from the expert interviews classified it as difficult to detect. How this discrepancy arises is unclear. TOCTOU errors can be grouped in the category concurrency errors. Knowledge about the order of execution is needed, which is not present in a static analysis but in a dynamic analysis, to detect concurrency errors.

Concerning the formats for error use cases the participants reported several different formats. CVEs, CWEs, source code snippets and SARIF[5] (static analysis interchange format). The responses CVE and CWE seem reasonable in response to this question as they both follow a defined scheme and thus could be used as a format for error use cases. The answer source code and SARIF on the other hand do not really fit for an error use case format. The term source code is very general and therefore is not suited for a general format for error use cases. Also, the response SARIF does not fit well as it is an open standard for the output of SAST tools to compare results across different tools. The standard was proposed by GrammaTech[6] and adopted by OASIS[7] (Organization for the Advancement of Structured Information Standards).

[5] https://docs.oasis-open.org/sarif/sarif/v2.0/sarif-v2.0.html
[6]https://blogs.grammatech.com/using-sarif-to-extend-analysis-of-sast-tools
[7]https://www.oasis-open.org/

6.4.5 ML and SAST Tools

In the end of the online survey the participants were asked to provide requirements for a SAST tool with machine learning support. The responses coincided with responses that were given in the expert interviews. Both sides noted that such a tool should have a high true positive rate while having a low false positive rate and low false negative rate. Furthermore, they emphasized on the fact that such a tool should be fast ("run almost in real-time") and efficient to use. Another requirement mentioned in the survey and in expert interviews was the ability to detect real-world errors and not just artificial errors. One of the survey participants stated that the tool's scope should be clear to everyone using it, and it should be customizable. If this is not fulfilled, the tool would be useless.

7 Elicitation and Explanation of Error Use Cases with Descriptions

ML SAST tools should provide functionality that goes beyond current SAST technology. One important aspect is to elicit error use cases that are often difficult to handle by current SAST tools. In this section, some of such error use cases are presented. These use cases have been identified during the interviews, but also by means of searching the literature and the documentation of vulnerabilities. In particular, the **common vulnerabilities and exposures list (CVE)** was used, which systematically lists reported vulnerabilities of IT products. It was assumed that more recent vulnerabilities (in open-source software) could be a good starting point for the envisioned error use cases because otherwise, current SAST tools would already have detected them.

As further possible sources for the identification of error use cases, the **SEI CERT C Coding Standard**[8], books on SAST like “Secure Programming with Static Analysis” by Chess and West[2] as well as the **common weakness enumeration (CWE)**[9] were considered. The CWE is a hierarchy of common types of weaknesses in software and hence gives a taxonomic view of software vulnerabilities. The vulnerabilities are organized in a hierarchical order that groups them based on their properties.

Subsequently, some error uses cases that have been elicited during the study (e.g., expert interviews and an online survey as well as the literature review) are briefly described. At the beginning of each problem description, a reference to one or more CWE entries is provided in the highlighted boxes shown below. Here, each CWE class is referenced by its unique ID as well as that of its base class, denoted in brackets. The base classes will be referenced again later in the ensuing chapters. CWE entries are closely related to the described error use case. The relationship to CWE entries will also become apparent when the results of the literature mapping are discussed since, in several ML-SAST publications, CWE entries are directly referenced to describe the problem space, see Chapter 10.

7.1 void* Pointers

CWE-704 (CWE-664): **Incorrect Type Conversion or Cast**

The software does not correctly convert an object, resource, or structure from one type to a different type.

Pointers play a crucial role in software development with C/C ++. Pointers represent a reference to regions in the main memory. Since C/C ++ allows programming that is very close to the hardware and access to memory areas can take place unchecked (low-level programming), static code analysis is dependent on capturing the context. This topic is not intended to be exhaustively discussed in this context. It is just about conveying a basic understanding of how pointers work and what relevance they have in relation to a static code analysis supported by machine learning. In the following, a specific problem related to potentially unsafe usage of pointers and the problems SAST tool may have detecting such cases is discussed.

C/C++ allows a developer to define pointers of void* type, i.e., it is valid to declare pointers of an undetermined data type, see Listing 2.

```
void *void_pointer = <memory address>;
```

Listing 2 Pointer declaration of undefined type in the C / C ++ programming languages.

Void pointers, for example, are often used when memory areas are allocated dynamically with the C-functions `malloc` and `calloc`. Due to the fact that these functions return memory regardless of the type of data to be stored, the return value types of `malloc` and `calloc` must be of the unspecific type void*. For example, the assignment depicted in Listing 3 reserves 1024 bytes dynamically in memory.

[8] https://wiki.sei.cmu.edu/confluence/display/c/SEI+CERT+C+Coding+Standard
[9] https://cwe.mitre.org/

```
void *x = malloc(1024);
```

Listing 3 Memory is allocated using the malloc function, which returns a pointer of undefined type.

In C, however, it is not possible to directly access this allocated memory. One must first cast the pointer `x` to a specific type before using it, as shown in Listing 4. The same remarks would apply when char values are to be stored in dynamically allocated memory.

```
int *y = (int *) x;
y[0] = 1;
```

Listing 4 The void pointer variable x from Listing 3 is casted into a pointer of the integer type before being assigned to y, as mandated by the C programming language.

From this discussion, one can see which flexibility C/C++ brings with it and which problems may arise from this flexibility for static code analyses. It is difficult for a SAST tool to distinguish between a correct usage (as shown above) and a potentially dangerous case; void* references could in principle point to memory of any type.

It should be noted that pointers are handled slightly different in C++. C++ makes available specific operators that support correct pointer usage. Furthermore, memory is usually requested in C++ with the new operator (rather than `malloc()`) and must be freed later with the delete operator (rather than `free()` as in C). The problem of using void* pointers, however, exists in both programming languages.

7.2 Difficulty in Tracking Tainted Data

CWE-707: Improper Neutralization

The product does not ensure or incorrectly ensures that structured messages or data are well-formed and that certain security properties are met before being read from an upstream component or sent to a downstream component.

Often data that are under the control of an attacker – for example, through network or file access – are read into the internal data structures of a C program. Such **tainted data** are difficult to track through the program flow to security-critical sinks, such as, `strcpy`, `strcat`, `memcpy`, `malloc`, `copy_from_user`, or `readlink`. This taint tracking is particularly difficult if these tainted data are multiply copied into different internal structs, which are later processed in the program text through (multiple) dereferencing of pointers.

The example of the **Heartbleed bug**, which commercial static analyzers could not find at that time, falls into this category. In the following, the problems with the static code analysis of such tainted data are explained and illustrated with the help of this Heartbleed bug.

The Heartbleed bug occurred in the **OpenSSL software** in spring of 2014 (versions 1.0.1 to 1.0.1f). OpenSSL provides cryptographic services such as an implementation of the **TLS protocol** and is widely used in many software stacks (often under the hood without developers knowing). Furthermore, the Heartbleed bug is discussed in several scientific papers on static code analysis for C/C++ for motivating the corresponding approaches[36], [42], [60]. Due to its relevance and the inherent difficulty in detecting it, this bug will be discussed in more detail.

The Heartbleed bug concerned the former heartbeat feature of OpenSSL's TLS implementation, hence its name. The heartbeat feature is quite specific and helps to periodically check with a small chunk of data whether the connection between the communication client and the server is still alive. The description of the bug relies of that of Green[61]. For receiving the heartbeat message (in form of a record), the following data structure is used:

```
typedef struct ssl3_record_st{
  /*r */  int type;               /* type of record */
  /*rw*/  unsigned int length;    /* How many bytes available */
  /*r */  unsigned int off;       /* read/write offset into 'buf' */
  /*rw*/  unsigned char *data;    /* pointer to the record data */
  /*rw*/  unsigned char *input;   /* where the decode bytes are */
  /*r */  unsigned char *comp;    /* only used with decompression - malloc()ed */
  /*r */  unsigned long epoch;    /* epoch number, needed by DTLS1 */
  /*r */  unsigned char seq_num[8]; /* sequence number, needed by DTLS1 */
} SSL3_RECORD;
```

Listing 5 Structure used by OpenSSL upon the receival of a heartbeat message.

```
int dtls1_process_heartbeat(SSL *s) {
  unsigned char *p = &s->s3->rrec.data[0], *pl;
  unsigned short hbtype;
  unsigned int payload;
  unsigned int padding = 16; /* Use minimum padding */
/* Read type and payload length first */
  hbtype = *p++;
  n2s(p, payload);
  pl = p;

  // do something with the payload
  if(s->msg_callback)
    s->msg_callback(0, s->version, TLS1_RT_HEARTBEAT,
                      &s->s3->rrec.data[0], s->s3->rrec.length, s,
                      s->msg_callback_arg);

  if(hbtype == TLS1_HB_REQUEST) {
    unsigned char *buffer, *bp;
    int r;

    /* Allocate memory for the response, size is 1 byte message type,
     * plus 2 bytes payload length, plus payload, plus padding
     */
    buffer = OPENSSL_malloc(1 + 2 + payload + padding);
    //allocate all that memory without any checks
    bp = buffer;
    /* Enter response type, length and copy payload */
    *bp++ = TLS1_HB_RESPONSE;
    s2n(payload, bp);
    memcpy(bp, pl, payload);
    bp += payload;
    /* Random padding */
    RAND_pseudo_bytes(bp, padding);
    r = dtls1_write_bytes(s, TLS1_RT_HEARTBEAT, buffer, 3 + payload +
                            padding);
    // send the response back, even the stuff the attacker wasn't supposed
    // to see
    if (r >= 0 && s->msg_callback)
      s->msg_callback(1, s->version, TLS1_RT_HEARTBEAT,
                        buffer, 3 + payload + padding,
                        s, s->msg_callback_arg);
    OPENSSL_free(buffer);
    if (r tlsext_hb_seq){
      dtls1_stop_timer(s);
      s->tlsext_hb_seq++;
      s->tlsext_hb_pending = 0;
    }
  }
  return 0;
}
```

Listing 6 The code section responsible for the Heartbleed bug. The affected lines are highlighted in bold letters.

A static analysis must recognize that the `data` entry in struct `SSL3_RECORD` is under the control of an attacker; a manual code review could use the comment `/* pointer to the record data */` to understand this relation. A static analysis, however, does not have access to this information.

The data that have been read from the network then flow into the function `dtls1_process_heartbeat` via the pointer `s` as shown in Listing 6, which presents the code vulnerable to the Heartbleed bug.

The analysis must then conclude that `unsigned char *p = &s->s3->rrec.data[0]` uses tainted data (considering the double dereference `s->s3->rrec.data`). On calling the macro `n2s(p, payload)`, the length of the data to be replayed in the heartbeat response by the server is stored in the variable `payload`. This variable must then be marked "tainted" by a static analysis tool. Finally, `payload` is used in the following call (critical sink):

```
memcpy(bp, pl, payload);
```

Then an attacker can control the length of the data that `memcpy` writes from buffer `pl` in buffer `bp` (please note that the third parameter of `memcpy`:

```
void *memcpy(void *dest, const void *src, size_t count)
```

is the number of the bytes to be copied). Because of this read overflow, an attacker can read sensitive data from the server's main memory, such as TLS master secrets or private keys, and let them send back to her via the call:

```
s->msg_callback(1, s->version, TLS1_RT_HEARTBEAT, buffer,
                    3 + payload + padding, s,
                    s->msg_callback_arg);
```

7.3 Integer Overflows Related to Tainted Data

CWE-190 (CWE-682): **Integer Overflow or Wraparound**

The software performs a calculation that can produce an integer overflow or wraparound, when the logic assumes that the resulting value will always be larger than the original value. This can introduce other weaknesses when the calculation is used for resource management or execution control.

Integer overflows can be induced by data that are under the control of an attacker (tainted data). The following example code, stemming from a paper by Yamaguchi et al., illustrates the problem[36]:

```
if (channelp) {
  /* set signal name (without SIG prefix) */
  uint32_t namelen = _libssh2_ntohu32(data + 9 + sizeof("exit-signal"));
  channelp->exit_signal = LIBSSH2_ALLOC(session, namelen + 1);

  // [...]

  memcpy(channelp->exit_signal, data + 13 + sizeof("exit_signal"),
namelen);
  channelp->exit_signal[namelen] = '\0';
}
```

Listing 7 Exemplary code segment, illustrating an integer overflow vulnerability [36].

Using the function call `libssh2_ntohu32()` the variable `namelen` is written and at the same time under external control. With the help of the call:

```
channelp->exit_signal = LIBSSH2_ALLOC(session, namelen + 1);
```

the memory for the variable `channel->exit_signal` is allocated on the heap, namely, of the size `namelen + 1` bytes. If an attacker now assigns the largest numerical value to this 32-bit-wide variable `namelen` can hold (from now on referred to as `UINT_MAX`), then, an integer overflow occurs such that `namelen + 1` evaluates to the value 0. This means that zero bytes are allocated. At the same time, the `memcpy()` call:

```
memcpy(channelp->exit_signal, data + 13 + sizeof("exit_signal"), namelen);
```

copies `UINT_MAX` bytes into the heap location `channelp->exit_signal`, which obviously leads to a heap memory overflow as only zero bytes have been allocated. It is crucial for this kind of error that memory is allocated by an integer variable that is under external control.

7.4 Array Boundary Errors

CWE-190 (CWE-664): **Improper Restriction of Operations within the Bounds of a Memory Buffer**

The software performs operations on a memory buffer, but it can read from or write to a memory location that is outside of the intended boundary of the buffer.

Array boundary errors occur when accesses take place outside the allocated area of the array. If an attacker succeeds in controlling array indices, then it is possible for him to read or write to memory areas outside the array, depending on the vulnerable software.

```
bool WDA_TxPacket(void * wma_context, void * tx_frame, eFrameType frmType,
                  tpPESession psessionEntry) {
  tp_wma_handle wma_handle = (tp_wma_handle)(wma_context);
  int32_t is_high_latency;
  u_int8_t downld_comp_required = 0;
  tpAniSirGlobal pMac;
  ol_txrx_vdev_handle txrx_vdev;
  u_int8_t vdev_id = psessionEntry->smeSessionId;

  if (NULL == wma_handle) {
    printf (" wma_handle is NULL \n ");
    return false;
  }
  pMac = (tpAniSirGlobal) vos_get_context(VOS_MOD_ID_PE,
            wma_context->vos_context);

  if(!pMac) return false;
  if(frmType >= HAL_TXRX_FRM_MAX) return false;
  if(!((frmType == HAL_TXRX_FRM_802_11_MGMT) ||
       (frmType == HAL_TXRX_FRM_802_11_DATA)))
    return false;
  txrx_vdev = wma_handle->inaterfaces[vdev_id].handle;
  if(!txrx_vdev) return false;
  if(frmType == HAL_TXRX_FRM_802_11_DATA) {
      adf_nbuf_t skb = (adf_nbuf_t) tx_frame ;
    adf_nbuf_t ret = ol_tx_non_std(txrx_vdev, ol_tx_spec_no_free, skb);

    if(ret) {
      // do something
    }

    is_high_latency = wdi_out_cfg_is_high_latency(
                            txrx_vdev->pdev->ctrl_pdev);
    downld_comp_required = is_high_latency && tx_frm_ota_comp_cb;
  }

  if(downld_comp_required) {
    // do something
  }

  return true;
}
```

Listing 8 An out of bounds error in Qualcomm WiFi driver Qcald-2.0. The critical code sections are marked bold.

Xiao et al. discuss a typical and more complex array bounds error of the Qualcomm WiFi driver Qcald-2.0, which is also installed on Android devices[62] (see Listing 8, with the critical code lines marked bold).

The function `WDA_TxPacket` has a fourth parameter `tpPESession`, which points to session information. The session ID, being accessible via `psessionEntry->smeSessionId`, may be under control of an attacker. Consequently, this attacker can also control the index of the array access `wma_handle->interfaces[`**`vdev_id`**`]`. The variable `txrx_vdev`, which is subsequently used in the code, may be undefined.

A static analysis must conclude that (1) `psessionEntry->smeSessionId` is under control of an attacker (i.e., it is tainted), (2) no additional security check is performed on this data ("sanitization") and (3) `wma_handle->interfaces[`**`vdev_id`**`]` is the critical array access. Only conservatively marking array accesses is not a solution because then the false-positive rate would become excessively high – a useful static analysis must be as precise as possible.

The problem might become even more complex if integer arithmetic is involved in accessing arrays as also discussed in the conducted expert interviews. Here, one of the interviewees gave the following example of vulnerable code that is usually hard to detect for static code analyzers:

```
void main() {
   int *s = malloc (300000);
   if (s != 0) {
     s[25] = 3;
     for (int i = 0; i < 600; i++) {
       s[i*i] = 5;
     }
     s[360000] = 10;
   }
}
```

Listing 9 Code example comprising an out of bounds memory access vulnerability that according to one of the interviewees from the expert interviews in Section 5 is hard to detect using SAST tools.

In this code excerpt, two lines are marked bold that contain array out of bounds errors. The second one `s[360000] = 10`; can in principle be detected using techniques like constant propagation, i.e., to calculate constant expressions statically at compile or analysis time. Despite being theoretically feasible, some tools seem to have problems even with this code. The second example is hard to identify for static code analysis because it involves integer arithmetic, in this case `i*i` for the array access. Then the analysis must be able to conclude that this expression can assume values larger than the array size. This means it must calculate the possible values of `i*i` and in particular its limits. Such analyses are possible in some specific case but are in general beyond current static code analysis technology.

7.5 Null Pointer Access

CWE-476 (CWE-710): **NULL Pointer Dereference**

A NULL pointer dereference occurs when the application dereferences a pointer that it expects to be valid, but is NULL, typically causing a crash or exit.

```
int cifs_close(struct inode *inode, struct file *file) {
  cifsFileInfo_put(file->private_data);
  file->private_data = NULL;

  /* return code from the ->release op is always ignored */
  return 0;
}
```

Listing 10 Exemplary code snippet, depicting a null pointer dereference.

Null pointer accesses, i.e., the dereferencing of null pointers, are another common class of security holes in C/C ++ code. The following vulnerability manifesting in the Linux kernel illustrates this problem, see Listing 10[62].

In this specific case `file->private_data` can be null and at the same time might be under control of an attacker. The statement `cifsFileInfo_put(file->private_data)` then uses `file->private_data` without any null pointer check.

In contrast, the following example from the same code base is a false positive[62]:

```
static int ubifs_dir_release (struct inode *dir, struct file *file){
  kfree(file -> private_data);
  file->private_data = NULL;
  return 0;
}
```

Listing 11 A similar, but non-vulnerable example showcasing a pointer dereference.

The program code in Listing 11 looks similar to the erroneous code shown above. The API call `kfree()` itself, however, already comprises null pointer checks.

7.6 Format String Errors (With Tainted Data)

CWE-134 (CWE-664): **Use of Externally-Controlled Format String**

The software uses a function that accepts a format string as an argument, but the format string originates from an external source.

C functions such as `printf`, `sprintf` or `vsprintf` expect, among other things, a format string as a parameter that formats the input according to the format string specification. It is problematic if the format string is not transferred as a constant and is at least partially made up of input data. Chess and West give the following example (FTP daemon wuftp)[2]:

```
while(fgets(buf, sizeof buf, f) {
  lreply(200, buf);
}


void lreply(int n, char *fmt, …) {
  char buf[BUFSIZ];

  // [...]

  vsnprintf(buf, sizeof buf, fmt, ap);
}
```

Listing 12 A code sample containing a format string vulnerability that may be triggered through tainted data.

The format string `fmt` used in the statement `vsnprintf(buf, sizeof buf, fmt, ap)` is assumed to be under the control of an attacker. Then she can manipulate memory areas in the context of the call stack of `vsnprintf` with a cleverly crafted definition of the format string and the input to `vsnprinf`[63].

Chess and West propose several rules to define format strings more securely[2]. Basically, format strings should be constant. If this is not possible, then one should select from a set of predefined format strings and if that again is not possible, then the input for the format string should be checked more thoroughly. Concerning this kind of error use case, an ML-based approach could learn **benign behavior** of format strings.

7.7 String Termination Errors

CWE-170 (CWE-707): **Improper Null Termination**

The software does not terminate or incorrectly terminates a string or array with a null character or equivalent terminator.

Another class of typical C /C ++ problems are string termination errors. The following example shows the problem[2]:

```
void badReadlink1(char *path) {
  char buf[PATH_MAX];
  int ret = readlink(path, buf, PATH_MAX);
  printf(”file is: %s\n”, buf);
}
```

Listing 13 Example of a string termination vulnerability.

The C function `readlink`, which returns the actual file name of a symbolic link in the variable `buf`, does not use null termination, i.e., `buf` is not null-terminated. If it will be used later, e.g., as a parameter of `strcpy` or `strcat`, a buffer overflow is possible if no additional checks are performed.

7.8 Missing Security Checks for Critical Operations

CWE-693: **Protection Mechanism Failure**

The product does not use or incorrectly uses a protection mechanism that provides sufficient defense against directed attacks against the product.

Often isolated **security checks** are **missing** in the code, which are supposed to safeguard security-critical operations. These can be **authorization checks**, but also conditions under which critical C APIs, such as `malloc`, `strcpy` and `memcpy`, may be called.

The following example (taken from the paper by Yamaguchi et al.) explains the point, see Listing 14 [64]:

```
int foo(char *user, char *str, size_t n) {
  char buf[BUF_SIZE], *ar;
  size_t len = strlen(str);

  if(!is_privileged(user)) return ERROR;
  if(len >= BUF_SIZE) return ERROR;

  memcpy(buf, str, len);
  ar = malloc(n);

  if(!ar) return ERROR;

  return process(ar, buf, len);
}
```

Listing 14 Example of a critical operation that must be guarded by a number of security checks.

Each of the if conditions that are marked bold represents a security-relevant check. The first one is an authorization check, the second one checks the string length needed for the `memcpy()` call, and the third check ensures that no null pointer is used in the succeeding call of the `process()` function. If one of these checks is missing, security problems may arise later in the program.

7.9 Unsafe Use of Random Number Generators

CWE-338 (CWE-330): **Use of Cryptographically Weak Pseudo-Random Number Generator (PRNG)**

The product uses a Pseudo-Random Number Generator (PRNG) in a security context, but the PRNG's algorithm is not cryptographically strong.

A typical problem in C code (often e.g., occurring in IoT system software) is the insecure use of **random number generators for cryptographic operations** (e.g., for generating keys, initialization vectors and nonces). The following problems can appear when cryptographic randomness is needed:

- An insecure random number generator is used, such as `rand`, `srand`, `srand48`, `drand48`, `lrand48`, `random`, and `srandom`.
- The cryptographic random number generator is seeded with too low entropy.
- Bad entropy sources are employed.

Static analysis often tends to flag the use of the aforementioned insecure random number generator API calls unconditionally. However, there are many situations in which no cryptographically strong randomness is needed – e.g., if a user interface element is to be displayed randomly. Many false positives are produced by this conservative approach of static analyzers such that the analysis might be too imprecise here.

ML-based approaches could help to learn whether randomness is used in a security-critical context or not. In addition, ML could be applied to learn if adequate entropy sources are used.

7.10 Information Flow Through Uncleared Main Memory Data Structures

CWE-226 (CWE-664): **Sensitive Information in Resource Not Removed Before Reuse**

The product releases a resource such as memory or a file so that it can be made available for reuse, but it does not clear or "zeroize" the information contained in the resource before the product performs a critical state transition or makes the resource available for reuse by other entities.

Security-critical data, such **as encryption keys or credentials, must be cleared from main memory** when a new value is assigned to such a variable. Otherwise, sensitive information may be leaked to an attacker. The following example, taken from a publication by Ahmadi et al.[65] demonstrates this problem, see Listing 15:

```
// Missing ‘OPENSSL_clear_free’
ec->key = key;
ec->keylen = keylen;
```

Listing 15 Exemplary code section depicting a data structure containing confidential information. By mistake, the fields have not been cleared before being assigned a new value.

In the OpenSSL library, the array `ec->key` is not cleared before the new value key is assigned, i.e., the call `OPENSSL_clear_free()` is missing. Usually, it is not easy to detect such problems because a static analysis must find out that those variables are security-critical and contain sensitive information.

7.11 Insecure Origin of Cryptographic Keys and Initialization Vectors

CWE-1204 (CWE-664): **Generation of Weak Initialization Vector (IV)**

The product uses a cryptographic primitive that uses an Initialization Vector (IV), but the product does not generate IVs that are sufficiently unpredictable or unique according to the expected cryptographic requirements for that primitive.

CWE-321 (CWE-693): **Use of Hard-coded Cryptographic Key**

The use of a hard-coded cryptographic key significantly increases the possibility that encrypted data may be recovered.

When encryption is used, the question of the **origin of the key or the source of the key material** often arises. The following code example, taken from the Wiki of the OpenSSL library, demonstrates the problem[10]:

```
int main (void) {
  /* A 256 bit key */
  unsigned char *key = (unsigned char *)"01234567890123456789012345678901";
  /* A 128 bit IV */
  unsigned char *iv = (unsigned char *)"0123456789012345";
  /* Message to be encrypted */
  unsigned char *plaintext = (unsigned char *)"The quick brown fox jumps
over the lazy dog";

  /*
   * Buffer for ciphertext. Ensure the buffer is long enough for the
   * ciphertext which may be longer than the plaintext, depending on the
   * algorithm and mode.
   */
  unsigned char ciphertext[128];
  /* Buffer for the decrypted text */
  unsigned char decryptedtext[128];
  int decryptedtext_len, ciphertext_len;

  /* Encrypt the plaintext */
  ciphertext_len = encrypt(plaintext, strlen ((char *)plaintext), key, iv,
ciphertext);
  /* Do something useful with the ciphertext here */
  printf("Ciphertext is:\n");
  BIO_dump_fp (stdout, (const char *)ciphertext, ciphertext_len);
  /* Decrypt the ciphertext */
  decryptedtext_len = decrypt(ciphertext, ciphertext_len, key, iv,
decryptedtext);

  /* Add a NULL terminator. We are expecting printable text */
  decryptedtext[decryptedtext_len] = '\0';
  /* Show the decrypted text */
  printf("Decrypted text is:\n");
  printf("%s\n", decryptedtext);

  return 0;
}
```

Listing 16 A code section, containing cryptographic operations that have not been initialized properly as the IV and symmetric key are hard coded.

The call

```
encrypt(plaintext, strlen ((char *)plaintext), key, iv, ciphertext)
```

uses a hard-coded **symmetric key and initialization vector (IV)**, see the first two bold lines. Even if this case is simple, it is generally difficult for static code analyses to trace back the origins of keys and IVs, specifically, if keys and IVs are generated in a different part of the code and a sophisticated inter-procedural static analysis is necessary.

[10] https://wiki.openssl.org/index.php/EVP_Symmetric_Encryption_and_Decryption

7.12 Reuse of Initialization Vectors for Specific Operating Modes of Block Ciphers

CWE-1204 (CWE-664): **Generation of Weak Initialization Vector (IV)**

The product uses a cryptographic primitive that uses an Initialization Vector (IV), but the product does not generate IVs that are sufficiently unpredictable or unique according to the expected cryptographic requirements for that primitive.

Encryption modes that simulate stream ciphers, such as CTR, GCM, CFB, and OFB, must ensure that the initialization vector IV is not used more than once to generate the keystream. In this case, an attacker who has the plaintext and the ciphertext could reconstruct the keystream and decrypt further messages[66].

A static program analysis would have to reconstruct the IV generation and recognize that the IV is always freshly generated.

7.13 Memory Leaks

CWE-401 (CWE-664): **Missing Release of Memory after Effective Lifetime**

The software does not sufficiently track and release allocated memory after it has been used, which slowly consumes remaining memory.

Memory can be requested (allocated) dynamically at runtime and released again. For this, the functions `malloc` (for a certain number of bytes) or `calloc` (for an array of elements of a certain size) for allocation and free for memory release are available in C. In C ++, however, new and delete are used.

It is the responsibility of the developer to release the requested memory areas after they have been used. If this does not happen, the available memory becomes scarce over time. Unreleased memory is known as a **memory leak**. In extreme cases, this can lead to the failure of the software used or even the entire system (out of memory error - OOM).

How can memory leaks be found? **Automatic garbage collection (GC)** is available in many languages, and there are also libraries for C/C ++. However, this GC can easily be bypassed, and in certain system-related areas it even has to be.

In C++, the use of **smart pointers**, which has been around since the first half of the 1990s, provides protection against memory leaks. Allocated memory is automatically released when there is no one left who can access it. Intelligent pointers can also alleviate the problem of **dangling pointers** as they prevent the memory from being released until an object is no longer used.

In the course of time, various C++ standards have introduced new linguistic constructs with which intelligent pointers can be used (`std::shared_ptr`, `std::make_shared`, `std::unique_ptr`, `std::make_unique`), and it is strongly recommended that these are preferred to the keyword `new` in as many situations as possible. The use of `new` could be found through static code analysis, whereby ML could possibly provide support with the evaluation.

Another approach is to check the source code for formal correctness. When developing complex software, this option is certainly more of theoretical nature as it is very time-consuming and requires extensive expert knowledge.

The possibly incorrect **manual management of the memory** can be determined with suitable tools at runtime. With tools like *memcheck* from *Valgrind*, it is possible to detect memory leaks in a running program, but this results in serious performance losses (typically losses of around 80%).

In the area of static code analysis, different tools try to address this problem. For example, *Cppcheck* claims to be able to detect memory leaks in program code. In certain situations, static code analysis can have advantages over graphs made up of references and memory areas, e.g., if it can be seen that allocated memory areas are never accessed for reading.

7.14 Dangling Pointers – Use-After-Free

CWE-825 (CWE-664): **Expired Pointer Dereference**

The program dereferences a pointer that contains a location for memory that was previously valid, but is no longer valid.

CWE-416 (CWE-664): **Use After Free**

Referencing memory after it has been freed can cause a program to crash, use unexpected values, or execute code.

As already mentioned, manual memory management in C / C ++ also leads to further problems. Pointers that have never been initialized or that point to memory that has already been released. The resulting behavior is unpredictable. It is possible that when an invalid pointer is accessed in this way, not even a runtime error occurs because the memory has not actually been released by the runtime system. However, it is also possible that the memory has already been assigned to the same process again, but in a different context. If the memory is no longer allocated to the process when it is accessed and a runtime error occurs due to a protection violation (access violation), a (user space) program is usually terminated. However, if such errors occur in the kernel context, this can affect the operating system. Due to this unpredictability, such errors are often not reproducible, and their cause is accordingly difficult to identify.

Not only do they affect the stability of software and the operating system, but they also lead to security gaps, as they can be exploited by attackers. Therefore, there is an entry in the Common Weakness Enumeration (CWE), namely CWE-416[67], for the problems of the **dangling pointers** and the **use-after-free**.

The following examples show a dangling pointer and a use-after-free in the context of a C program and a C++ program:

```
int main(void) {
   int* a;
   /* allocate memory. */
   a = malloc(sizeof(int));
   *a = 10;
   /* … */
   free(a);
   /* … */
   *a = 20;
}
```

Listing 17 A section of a C program, demonstrating the problematic nature of dangling pointers.

```
int main(void) {
   int *intPointer = new int;
   *intPointer = 10;
   /* … */
   delete intPointer;
   /* … */
   *intPointer = 20;
   return 0;
}
```

Listing 18 Another example depicting the access of a dangling pointer. This time in the C++ programming language.

Here, too, the occurrence of these errors can be detected during runtime with tools such as *memcheck* from *Valgrind*. Finding it by means of a static code analysis is inherently difficult since the failure to initialize a pointer or the release of allocated memory can be located in the code at a point that is far away from the use of this pointer.

The fact that the value of a pointer can be assigned to another pointer (alias) also makes traceability more difficult, see Listing 19:

```
int *p, *q;
p = malloc(sizeof(int));
q = p;
free(p);
/* From here on p and q are dangling pointers. */
/* Every access to q is a Use After Free */
```

Listing 19 Lastly, dangling pointers may be hard to detect, as they may be masked through aliasing.

As already described, newer C++ standards with the use of smart pointers offer a means of mitigating the dangling pointers, since they make it possible to prevent memory release until an object is no longer used. A static analysis approach for C++ would therefore be to examine the occurrences of the keyword `new` more closely.

In a paper from 2020, Kailong Zhu, Yuliang Lu and Hui Huang present a method for identifying security gaps caused by use-after-free[68]. However, this method is not applied to source code, but rather to binary code.

7.15 Race Conditions

CWE-362 (CWE-691): **Concurrent Execution using Shared Resource with Improper Synchronization**

The program contains a code sequence that can run concurrently with other code, and the code sequence requires temporary, exclusive access to a shared resource, but a timing window exists in which the shared resource can be modified by another code sequence that is operating concurrently.

CWE-366 (CWE-691): **Race Condition within a Thread**

If two threads of execution use a resource simultaneously, there exists the possibility that resources may be used while invalid, in turn making the state of execution undefined.

More complex **concurrency problems via multithreading or shared memory** are difficult to find with static program analyzes automatically, since concurrency would have to be modeled. The following code example, taken from the description of CWE-366, demonstrates the problem, see Listing 20 [69]:

```
int foo = 0;
int storenum(int num) {
  static int counter = 0;
  counter++;
  if (num > foo) foo = num;
  return foo;
}
```

Listing 20 A code sample, showcasing the problems associated with concurrency.

The static int variable `counter` is modified here without any lock. In a multi-threading application this value would then be non-deterministic. A contrived static program analysis would have to model this kind of multi-threading, which is difficult to do. Graf introduces in his dissertation such an approach for Java[31].

7.16 Application-Specific Errors

CWE-670 (CWE-691): **Always-Incorrect Control Flow Implementation**

The code contains a control flow path that does not reflect the algorithm that the path is intended to implement, leading to incorrect behavior any time this path is navigated.

Logical errors are usually difficult to detect through static program analysis. An example of such a logical flaw is the known ShellShock vulnerability in the Bash shell (CVE-2014-6271).

A largely unknown Bash feature is used here, in which function definitions are also allowed in environment variables (for the shell). These function definitions can then be exported from Bash to a Bash child process. The function definitions are initially only parsed and not executed.

If one considers the following Bash function definition (anonymous function without name) in the environment variable `testbug` as shown in Listing 21

```
testbug = () { :;}; echo VULNERABLE
```

Listing 21 Example shell script that triggers the so-called ShellShock vulnerability in affected versions of the Bash shell (CVE-2014-6271).

then the Bash shell first parses the definition `() { :;}`. At the same time, the subsequent Bash command `echo VULNERABLE` is executed. A remote attacker managing to infiltrate the environment variable, e.g., on a web server, may gain the ability to execute arbitrary commands or possibly even control over the server entirely.

The fix of the aforementioned problem includes adding the following if statement in the corresponding function `parse_and_execute`:

```
if((flags & SEVAL_FUNCDEF) && command->type != cm_function_def)...
```

Listing 22 The fix in the C code of the Bash shell, for the so-called ShellShock vulnerability, depicted in Listing 21.

This fix avoids that parsing of functions `(flags & SEVAL_FUNCDEF)` and executing commands `(command->type != cm_function_def)` are performed at the same time – this had led to the vulnerability mentioned above. Static analyses can hardly detect such application-specific security issues as this would require an advanced analysis to understand the application logic.

7.17 Pointer Casts to Types with Stricter Alignment

CWE-843 (CWE-664): **Access of Resource Using Incompatible Type**

The program allocates or initializes a resource such as a pointer, object, or variable using one type, but it later accesses that resource using a type that is incompatible with the original type.

The **casting of pointers**, either explicitly or via a void pointer, in pointers of a data type with a stricter alignment leads to undefined behavior according to the C-standard ISO-9899: 2011. Depending on the architecture, access to a memory area via such a pointer can lead to program crashes or loss of information. The following example illustrates the problem[70], see Listing 23:

```
#include <assert.h>
void func(void) {
  char c = 'x';
  int *ip = (int *)&c; /* This can lose information */
  char *cp = (char *)ip;
  /* Will fail on some conforming implementations */
  assert(cp == &c);
}
```

Listing 23 A code example, highlighting the problem of the potential loss of data, when a pointer is being casted into a datatype with stricter alignment.

8 Evaluation Criteria for ML-SAST Approaches

As can be inferred from the expert interviews and from the online questionnaire in Chapters 5 and 6, evaluating the suitability of the individual solutions for ML-SAST involves two dimensions. Concerning the *qualitative dimension*, a criterion that was often named by the experts as well as the users was the ability to be able to explain why the tools predicted a given code section as vulnerable. In contrast to that, there is the *quantitative dimension*. Here one criterion frequently stated was that the tools should offer a low number of false alerts. Whereas the latter criterion may be easily expressed in numbers, the former cannot be expressed that way. It is possible, however, to address such qualitative attributes through principles.

Based on the findings from the expert interviews and the online questionnaire, as well as the literature on the topic, this chapter explores different means for the evaluation of machine learning-based SAST solutions. As this study aims to provide a sound basis in the possible preparation of a concrete SAST tool based on ML technologies, it is important to investigate such evaluation devices, in order to provide the necessary guidance.

8.1 Qualitative Criteria

The interviews and the questionnaire in Chapters 5 and 6 have resulted in a number of qualitative criteria concerning ML-SAST solutions, deemed highly important by users and experts alike. Picking up on these demands, this section aims to unite and structure the individual statements by the experts and users, consulting the Ethics Guidelines For Trustworthy AI, provided by the Independent High-Level Expert Group (HLEG) on Artificial Intelligence of the European Commission[8].

8.1.1 Transparency

The HLEG uses transparency as an umbrella term that comprises the three aspects of explainability, traceability and communication.

Explainability refers to the ability to trace back and understand the process of how the model derived its prediction from a given input. Furthermore, the HLEG also states that the explainability not only involves these technical aspects, but also the human decisions involved. This, however, is out of the scope of this study so that whenever explainability is mentioned hereafter, it is purely referring to the aforementioned technical aspects. Examples where the decisions made by a model cannot be traced back are sometimes referred to as black-box AI[5], [8]. Emerging from a funding opportunity report by the DARPA, there also often are inherit tensions between the predictive performance of models and their explainability[71]. Due to the high complexity of such models, it is practically impossible to derive explanations over the entirety of the input space. For much more simple systems, however, it is possible to do so[72]. The criterion of explainability was mentioned multiple times by the users and in particular by the experts during the interviews and the online questionnaire. This may not come as much of a surprise since both groups also emphasized over- and underestimation of SAST tools as a major detriment, limiting their usefulness. Then, the situation is aggravated by the opaqueness of the tools. Not knowing as to why the tool detected a vulnerability, makes it hard for users tell real vulnerabilities apart from false alerts. If the tool then also shows a tendency to overestimate the likelihood of vulnerabilities, the users may be inclined to underestimate the likelihood of vulnerabilities, due to fatigue. Subsequently, real vulnerabilities may be accidentally dismissed as false alerts, directly affecting the security. The aspect of explainability and partial remedies to this end will be discussed in much more detail in the ensuing Chapters 11 and 12.

Traceability, according to the HLEG, refers to everything in terms of data that influences the decisions the models make and directly benefits the explainability. This involves, for instance, the datasets used for training and the accompanying processes, or the algorithms employed to this end. These aspects must be well documented to enable post-mortem analysis in cases where the model fails[8]. This was also mentioned by the users and experts, albeit often not directly. Regarding that matter one of the users stated the requirement: “That the developers can actually give a real explanation [sic] for how it works […]”. This highlights the importance of open-source software in the context of ML-SAST. However, such information is usually regarded as intellectual property by the developers of the tools and a highly reproducibility.

Communication, while not entirely unimportant, is certainly not as significant in the context of ML-SAST, as explainability and traceability. This requirement demands that users must be informed of the fact that they interact with AI and not in fact a human agent. Furthermore and more importantly, the limitations and capabilities of the system must be clearly communicated to the user[8]. This requirement was also discussed during the questionnaire by one of the participants. Providing this kind of information is important for two reasons. First, if the users of ML-SAST solutions are not informed about their limitations and capabilities, they may not be able to determine usage scenarios, where the application of such tools is appropriate. This may result in frustration and a low acceptance of such tools. Secondly, if the users are unable to correctly assess the limitations of the ML-SAST tools, then they may overestimate their capabilities, which may directly impact security.

8.1.2 Other Requirements

Aside from the requirements outlined by the European HLEG on AI, there are also other qualitative metrics frequently discussed in the literature.

Interpretability is often times referred to suggest that interpretability is the degree to which an observer can understand the cause of a decision[73]. It is sometimes used in reference to a stronger requirement than explainability. Whereas explainable models, or rather their predictions, may be explained using post-hoc methods, later discussed in Section 11.3, interpretable models themselves suffice for explaining its output[74]. There are few examples of algorithms that satisfy this definition. Examples comprise decision trees or sparse linear models for instance[75].

Robustness commonly refers to a model’s sensitivity towards small input changes that have drastic effects on the output and is often times difficult to determine as machine learning models are largely defined by their parameters, i.e., the weights and biases that are subject to an optimization algorithm. As a result of this, the characteristics of the model cannot be understood in its entirety, thus leaving the possibility of perturbations leading to unexpected predictions. In the context of ML-SAST, an adversarial user could intentionally try to leverage such perturbations in order to compromise the ability of a ML-based SAST tool to correctly detect vulnerabilities. For example, a vulnerability could be hidden from the tool through small syntactic changes, which do not affect the semantics however. An often-cited example by Goodfellow et al. from the domain of ML-based image recognition, illustrates the example. By adding miniscule amounts of noise to the input of a neural network-based image classifier, the authors were able to trick the model into misclassifying the image with very high confidence.

As can be seen in Figure 15, the input image originally depicts a panda that, although with low confidence, was correctly assessed as such by the model. After the addition of carefully selected perturbations, however, the model classified the image as depicting a gibbon, with very high confidence[76].

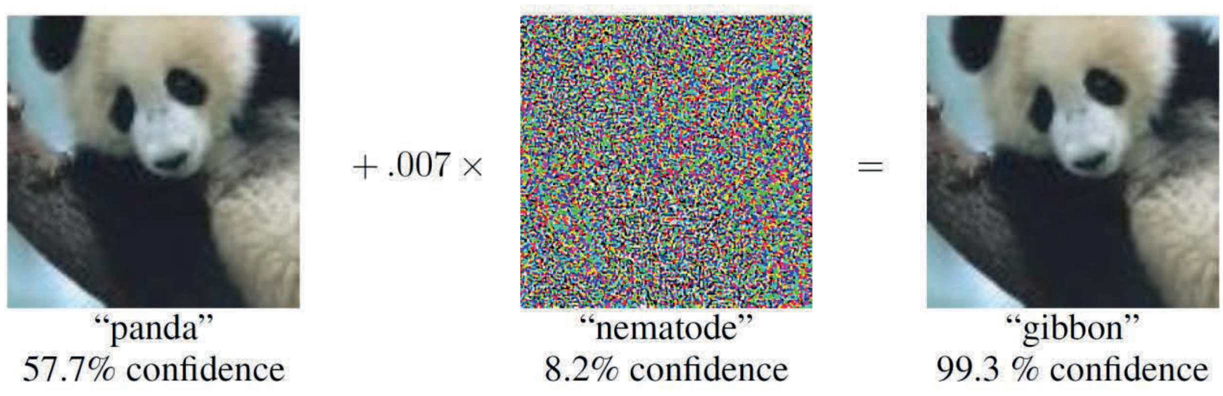


Figure 15 The addition of small perturbations to the input of an image classifier led to a gross misclassification.

8.2 Quantitative Criteria

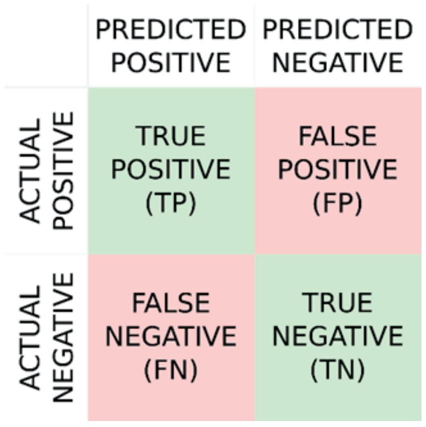
As discussed, aside from the qualitative requirements, there are also quantitative criteria by which individual ML-based SAST approaches may be evaluated and compared. Among these, especially the predictive performance of the solutions is of interest, as it shows how good it is at detecting vulnerabilities.

One of the most important aspects in terms of these criteria are metrics, used to determine the predictive performance of a machine learning model. They are usually derived from four core metrics, depicted in Table 6. At the example of ML-SAST, a binary classifier model may accept some arbitrary code samples as its input and based on these, makes a prediction on whether the sample belongs to the negative or positive class, i.e., the sample does not or does contain a vulnerable code pattern. Realistically, such model is not perfect and does not always come to the correct conclusion. If the model misclassifies a negative (benign) sample as positive (vulnerable) it is referred to as a **false positive**. Vice versa, when the model fails to detect an actually positive (vulnerable) sample, classifying it as negative (benign), it is referred to as a **false negative**. Consequently, the cases where the model correctly predicts a negative (benign) or a positive (vulnerable) as such, it is called a **true negative** or respectively a **true positive**. Extended to not only individual cases but a sequence of predictions over multiple samples, it is possible to calculate the corresponding rates at which a certain event occurs.

Table 6 Terminology used in confusion matrices to express the possible scenarios.

Type	***Description***	***Type***	***Description***
Positive	Number of real positive cases in data	Negative	Number of real negative cases in data
True Positive	Correctly predicted positive samples	True Negative	Correctly predicted negative samples
False Positive	Falsely predicted positive samples (type I error or underestimation)	False Negative	Falsely predicted negative samples (type II error or overestimation)

At the example of false positives, the rate at which the model makes this type of misclassification is determined by the ratio of the number of negative samples predicted as positive versus the total number of negative samples and referred to as the **false positive rate (FPR)**. Analogous to this, the **false negative rate (FNR)** is determined by the ratio of positive samples erroneously classified as negative versus the total number of positive samples. The same principle holds true for the **true negative rate (TNR)** and the **true positive rate (TPR)**, defined by the ratio of samples predicted negative and the number of actually negative samples and the number of positive predictions versus the number of actually positive samples, respectively. A concise and commonly used representation of these relations is the so-called confusion matrix, depicted in Figure 16.



	PREDICTED POSITIVE	PREDICTED NEGATIVE
ACTUAL POSITIVE	TRUE POSITIVE (TP)	FALSE POSITIVE (FP)
ACTUAL NEGATIVE	FALSE NEGATIVE (FN)	TRUE NEGATIVE (TN)

Figure 16 Graphical depiction of a confusion matrix.

Based on these four basic metrics other, more complex metrics may be derived that emphasize on different aspects. The choice of which is highly dependent on the intended application and the characteristics of the underlying problem. As for ML-SAST, a variety of **metrics** are consistently employed to quantify the predictive performance. Some are better suited for this task than others. One of the key characteristics in regard to SAST is the presence of a high level of **class imbalance**. By default, most of the source code in real-world applications is benign, with only a fraction posing a threat to security. In this light, some of the metrics commonly applied are not fit for this task, as they exhibit a tendency to cover bad performance. The remainder of this section elaborates on the most commonly used metrics in the literature concerning ML-SAST and their appropriateness to this end.

Accuracy, is defined as the ratio of correct predictions versus all predictions. As can be easily seen, due to the high number of benign code samples, a model that only ever classifies code as safe would still yield very high accuracy. Vice versa, the error rate equally does not properly reflect the characteristics of the underlying problem and should thus be avoided [78, p. 512]. Formally speaking the accuracy is defined as:

$$\mathrm{ACC} = \frac{\mathrm{TP} + \mathrm{TN}}{\mathrm{TP} + \mathrm{TN} + \mathrm{FP} + \mathrm{FN}}$$

Precision, sometimes referred to as the **positive predictive value (PPV)**, is another metric that is commonly used to assess the performance of ML-based SAST approaches. Arp et al. state that **precision** is a better-suited metric for tasks with high class-imbalance[43, p. 12]. The precision, however, only considers the system's ability to predict **true positive** samples and as such could be considered biased. Mathematically the precision is defined as:

$$\mathrm{PPV} = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FP}}$$

Markedness, is an unbiased alternative to the precision that does not ignore the number of true vulnerabilities left undetected, as it also takes the number of **false negatives** into account[77, p. 512]. The false negatives may be considered an important criterion when evaluating ML-based SAST approach and has also been mentioned during the interviews and the questionnaire.

$$\mathrm{MKN} = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FP}} + \frac{\mathrm{TN}}{\mathrm{TN} + \mathrm{FN}} - 1$$

Recall, is another term for the aforementioned TPR and poses a metric that has also been recommended by Arp et al. in order to assess the performance of ML-SAST approaches[43, p. 12]. It may be considered the counterpart to precision, used to determine the model's ability to correctly detect true positives.

$$\mathrm{TPR} = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FN}}$$

Informedness, is the pendant to the markedness. Similarly, to the markedness that considers the false negatives unlike the precision metric, the informedness also considers false positives that the recall does not. According to Antunes et al., this metric should be preferred over the arguably more popular **F-measure**[77, p. 512]. Unlike the recall metric it is thus unbiased. The informedness is defined as:

$$\mathrm{INF} = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FN}} + \frac{\mathrm{TN}}{\mathrm{TN} + \mathrm{FP}} - 1$$

F-Measure, often also referred to as F-score for F_1-score, is the harmonic mean of precision and recall. It is used particularly often in the literature on ML-SAST, but not necessarily the best choice. This is due to the fact that it penalizes those cases where precision and recall are far apart from each other. Given two models that show a similar TP, TN and FN, but differ vastly in FP, their F-scores would be very similar[77, p. 512]. In situations where false negatives matter however, as is arguably the case for ML-SAST, the metrics fails to provide sufficient means for distinguishing the predictive performance of the models in this regard. The F-measure is defined as:

$$\mathrm{F}_1 = 2 \cdot \frac{\mathrm{PPV} \cdot \mathrm{TPR}}{\mathrm{PPV} + \mathrm{TPR}}$$

Matthews correlation coefficient (MCC), is considered to be the metric of choice for evaluating ML-SAST approaches as it is an unbiased representation of the correlation between the predicted classes and the actual classes[43], [54], [78]. It is defined as the geometric mean of markedness and informedness, introduced before[77].

$$\frac{\mathrm{TP} \cdot \mathrm{TN} - \mathrm{FP} \cdot \mathrm{FN}}{\sqrt{(\mathrm{TP} + \mathrm{FP}) \cdot (\mathrm{TP} + \mathrm{FN}) \cdot (\mathrm{TN} + \mathrm{FP}) \cdot (\mathrm{FN} + \mathrm{FN})}}$$

ROC curves, is the abbreviation of receiver operating characteristic curves and as the name suggests, originate from signal processing. It is not a metric by itself, but rather the plot of the FPR and TPR under a varying classification threshold [78]. A perfect ROC curve would be equivalent to $\mathrm{f(x)} = 1$, whereas one equivalent to $\mathrm{f(x)} = 0$ would be a classifier that constantly predicts the wrong class. The worst case would be a curve defined by $\mathrm{f(x)} = \mathrm{x}$, equivalent to randomly picking classes. Such classifier would have no predictive capabilities at all. ROC curves may be useful in the context of ML-SAST, but only in the presence of adequate baselines[43, p. 12], [77, p. 508].

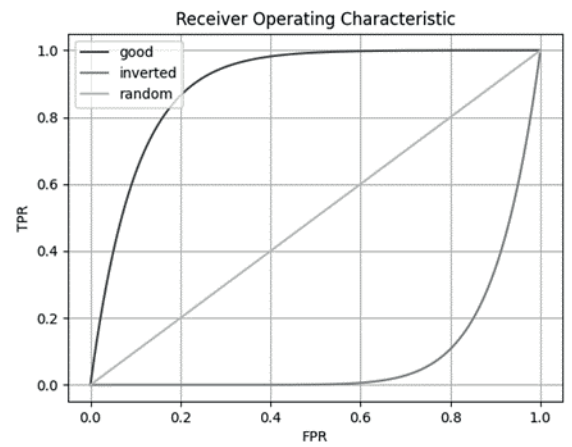


Figure 17 An exemplary depiction of a roc curve where the curves labeled "good", "inverted" and "random" are representative of a good classifier, one that constantly predicts wrong and one with no discriminative capabilities at all.

Area under Curve, refers to the area under the ROC curve and is thus representative of the discriminative performance of the classifier across all thresholds. It produces a single value between 1, a perfect classifier and 0, an inverted classifier, i.e., a classifier always making the wrong prediction.

Calculation of this metric is associated with high computational cost[79]. Depending on the application, this may be considered undesirable.

Despite some of the metrics mentioned being inappropriate in the context of ML-SAST, they should still be considered for evaluation tasks, as the F-measure may be considered the de facto standard across this field of research, see Chapter 10.

8.3 Other Criteria

The expert interviews as well as the survey in Chapters 5 and 6 have both shown that most of the experts and users were mainly concerned with the predictive performance. Here, specifically a low FPR and a high TPR were expressed as essential. For the sake of completeness, the computational complexity as a criterion to determine suitable algorithms for ML-SAST should not be left out. In that regard, some of the participants stated that to them a fast-performing solution would be indeed important. It must be also noted however, from the data gathered during the interviews and the questionnaire, it is unclear how the participants weigh these criteria. One of the participants stated on that matter that a ML-based SAST solution must be fast and potentially work in real time.

Arguably, this criterion is already implicitly provided, as the application of machine learning techniques should not pose a limitation in that regard. On the contrary, in some cases where there exists an algorithmic solution to a problem, but at a high computational cost, ML-based approaches can sufficiently approximate a solution at significantly reduced complexity. Rizi et al. for instance, were able to efficiently predict shortest paths in graphs in linear time, by employing machine learning techniques[80].

As for now, this criterion should be assigned a lower priority since a slower, but therefore transparent solution with adequate predictive performance ought to be considered preferable, especially for a prototypical solution.

On a related note, one of the participants of the survey remarked that an ML-based SAST approach must be well integrated into an integrated development environment. This criterion indeed touches upon the topic of computational complexity, as the model would likely have to constantly assess the code for vulnerable code patterns in the background. Another participant also expressed that it should be possible to run such solution offline, i.e., the model must be stored locally, not on a remote machine. Again, these criteria are much more related to architectural considerations of a solution that is ready for productive use and not a prototypical implementation that this study aims to provide a basis for.

9 Data in ML-Experiments: Types, Formats and Schemata

Beyond the background that this study aims to provide a sound basis for the prototypical development of ML-based SAST tools, the evaluation of different approaches through experiments depicts another important topic. During development, when searching for suitable algorithms or the best settings for the hyper parameters of a model, several alternatives may be investigated. These kinds of experiments require and generate data. It is thus important to organize these data not only to draw the right conclusions in terms of finding the best-performing solution, but also for the sake of documenting the process. As elaborated in Chapter 8, the requirement of traceability involves all data that accrues in the lifecycle of an ML-based SAST solution.

This section delves into this topic, discussing the different kinds of data potentially present during the development lifecycle of an ML-SAST approach. This involves benchmarks and the ground truth, experiment tracking, data formats and general ontologies to this end.

9.1 Benchmarks

Juliet is a collection of security test cases that exists for different programming languages, such as C#, Java, and C/C++. Its most recent issue is labelled 1.3 to remain consistent with the test suites for the C, C++ and Java programming languages.

The **OWASP** benchmark for security automation is a free and open test suite designed to evaluate the speed, coverage, and accuracy of automated software vulnerability detection tools and services. Without measuring these tools, it is difficult to understand their strengths and weaknesses and compare them to each other. Each version of the OWASP benchmark contains thousands of test cases that are fully runnable and exploitable, each of which maps to the appropriate CWE number for that vulnerability.

Code samples from real-world applications are another essential factor in benchmarking SAST approaches. Synthetic datasets used for benchmarking are oftentimes overly simplistic. For rule-based SAST solutions, this may not pose much of a problem; however, in the case of machine learning-based approaches that rely on representation learning, it is important to assert their ability to infer generalizable patterns. Furthermore, the simplistic examples found in synthetic datasets do not adequately represent the complexity of real-world vulnerabilities that may stretch across large distances. Potential approaches may thus be capable of detecting errors in synthetic data, where distances are short but fail to infer correlations over long distances in real-world data.

Several datasets are available that provide labeled samples from a variety of open-source projects that contain actual vulnerabilities that have accumulated during the software's lifetime. Chakraborty et al. proposed a dataset named **ReVeal**. The dataset comprises code samples mined from issue trackers of Debian Linux and the Chrome web browser. The authors provide the code samples in the form of the human-readable JSON file format. In addition to this, the authors also made a dataset available that consists of code samples from the QEMU and FFMpeg projects, known as the **Devign dataset** and equally in the form of a JSON file. The reveal dataset consists of 18,169 functions, of which 9.16% are vulnerable. The Devign dataset comprises a total of 22,361 samples, with 45.02% being vulnerable. The samples cannot be compiled using conventional compilers in both cases, as they are incomplete[81]. They can, however, be parsed by the Joern tool, for instance, which generates code property graphs and appears to be a popular choice in the landscape of ML-based SAST approaches[82], [83]. Lastly, there is another dataset, proposed by Kim et al., named **Draper VDisc**. The set comprises 1,27 million functions, mined from various open-source projects from GitHub. The labels for the set have been generated using an ensemble of conventional SAST tools so that a high amount of label noise is to be expected. The dataset is stored in the binary HDF5-format[84]. Russel et al. describe the inception of the dataset in a separate paper[46].

9.2 Type, Format and Metrics

The choice of the right metric plays an important role and is highly dependent on the application at hand. The metrics available for the evaluation of machine learning models were extensively discussed in Chapter 11. For the sake of brevity, they will thus not be further elaborated on in this section.

The results of the benchmarks should be stored in a human-readable but also machine-processible format. Possible options would be the **JSON or YAML file formats** as they allow for both and are easily exchangeable. Furthermore, these file formats are well integrated into the Python programming language, which is frequently employed in machine learning as there are various libraries available for this task[85]–[87].

In addition, the possibility to track results and possibly persist models and their parameters should be facilitated. **MLflow** is a free and open-source framework that constitutes an integrated solution to these problems. The platform allows tracking several metrics across a series of benchmark runs. These runs may then be aggregated into experiments. Moreover, artifacts such as models and training data may be persisted and shared between project contributors. Series of runs may also be exported as a simple comma-separated list and thus be easily converted into any desired format[88].

9.2.1 Name, Version and Description of Dataset

As datasets may grow large very quickly, they should be treated separately from benchmarks. Moreover, as there may be other uses for such data and to adhere to the terminology of MLflow, benchmarks should more generally be referred to as experiments. To render both types of data discernible, they must be denoted as either of type **experiment** or **dataset** by means of a designated **datatype** field.

Each dataset or experiment should be identifiable through a unique and readable name, e.g., *Draper VDisc*. Additionally, different versions of the same dataset may exist or the continuation of a previous experiment, necessitating an additional version number. Lastly, a field should be reserved for an optional user-defined description of the data. In addition to these fields, a hash may be generated over the version and name of the data defined prior to be easily distinguishable by machines.

9.2.2 Dataset Scheme

In the case of supervised learning, training and benchmarking datasets should be virtually indistinguishable, as they both form lists of labeled code samples. Consequently, there are no special considerations as to the purpose of the data that need to be made. These datasets are essentially lists of tuples, each of which consists of an input value and a target value.

Some important **meta information** for each dataset needs to be conveyed in some way, mandating a designated **meta field**, pointing to a dictionary of such information. Input values may take a variety of formats, e.g., representations of graph structures, code snippets or intermediate representations of such. The storage format should consider this fact using a **type descriptor**, denoting the type of the input values. In the case of binary data, the input values must be base64-encoded, which must also be signified using a designated field.

The type of the targets may also vary, depending on the application at hand. For example, in the case of binary classification, Boolean values should suffice. However, in the case of a multinomial classification problem, any other datatype may represent each individual class, e.g., CWE error classes. The type of classification problem should thus be denoted in another field.

Other fields that should be included are the maximum length of the samples, the total number of samples, and the number of positive samples. These fields are binding, meaning that they must hold true over the entirety of the dataset. Lastly, the dataset itself, a list of tuples with a length of two, is indicated by a field named “data”.

```
{
    "datatype": "dataset",
    "name": STR,
    "version": STR,
    "description": STR,
    "meta": {
        "inputFormat": STR,
        "targetFormat": "bool"|"multi",
        "base64": BOOL,
        "numSamples": INT,
        "trueSamples": INT,
        "maxLen": INT
    },
    "data": {
        [x0, y0],
        [x1, y1],
        [x2, y2],
           ...
        [xn, ym],
    }
}
```

Listing 24 Proposal of a JSON schema for machine learning datasets in the context of SAST.

9.2.3 Experiment Scheme

```
{
    "datatype": "experiment",
    "name": STR,
    "version": STR,
    "description": STR,
    "meta": {
        "dataset": STR,
        "model": STR,
        "params": [
            STR,
            ...
            STR
        ],
        "metrics": [
            STR,
            ...
            STR
        ],
    },
    "runs": [{
        "name": STR,
        "params": [p0, p1, p2, ..., pn],
        "results": [m0, m1, m2, ..., mn]
    }]
}
```

Listing 25 Proposal of a JSON schema for machine learning experiments in the domain of static application security testing.

The scheme of experiments follows that of the datasets. First, the “datatype” field is indicated as type “experiment”, followed by the experiment's name, version, and user-defined description, as before for the datasets. The “meta” field, however, holds entirely different information. Here the dataset and model used for the experiment are referenced by its unique identifier and the hash value, respectively. In addition, a set of model parameters may be enumerated in an ordered list by the key of “params”. Equally, the metrics used for the experiment must be contained in an ordered list by the key of “metrics”. Lastly, the individual runs that comprise each experiment are contained in the field runs, an ordered list. Each run carries a name, the values of the parameters set for the run and the results for each metric. Both the parameters and the results must follow the order indicated by the “params” field of the meta-information. If some metric was skipped during a run, the field must be set to “none” or some other value, indicating its absence. It may not be left out, i.e., shortening the list.

9.2.4 Model Persistence and Tracking

The versioning and persisting of the machine learning models may be realized through MLflow. Nevertheless, the need to export and share models between different contributors or users of the models may exist. Most libraries for the Python programming language are capable of handling files of the **hierarchical HDF5-format**. Another format frequently used for the persistence of machine learning models is the **PKL-file format**.

MLflow offers support for a multitude of different file storage backends to host projects as well as the various iterations of the individual models. This may, however, require additional infrastructure.

9.3 Ontologies for Machine Learning

Aside from task-specific schemata, recent efforts have been to encapsulate the lifecycle of various machine-learning applications in **higher-order ontologies**. These approaches are much more generalizable and provide high levels of exchangeability and reproducibility. As such, they may be favorable over application-specific schemata. However, the research on these ontologies is still in its infancy, and none of the proposed solutions has been widely adopted yet. Nevertheless, for the sake of completeness, the existence of the efforts in the research field of machine learning should at least be made a note of. An example for such ontologies would be, for instance, the so-called **MEX vocabulary**, a **lightweight ontology** comprised of 402 unique classes. The MEX vocabulary extends the **PROV ontology** of the **world wide web consortium (W3C)** for **data provenance**. The authors plan the integration of the OpenML[11] platform as well as other tools, such as Weka[12][89]. Similar efforts are Exposé, OntoDM-core and DMOP, the latter two stemming from the data mining rather than the machine learning domain. The W3C itself has recently started work on the so-called **ML-Schema**, a schema meant to align the aforementioned ontologies in a lightweight schema[90].

[11] https://www.openml.org/
[12] https://www.cs.waikato.ac.nz/ml/weka/

10 Approaches to Conducting a Literature Mapping for ML-SAST

The systematic identification of highly relevant literature in the field of ML-SAST and the ensuing extraction of the used techniques require the screening of a large body of publications. It has been decided to conduct a **systematic mapping study** to facilitate this endeavor in combination with the aptly named **snowball sampling method**. Systematic mapping studies, as proposed by Petersen et al. are comprised of the five individual steps listed below:

1. Definition of research questions
2. Search for relevant literature
3. Revision of the literature
4. Creation of a classification scheme
5. Data extraction and synthesis

Petersen et al. perform the second step by manually searching scientific databases only[91, pp. 2–3]. Another iterative approach, however, has been proposed by Wohlin. Here, starting from a predefined set of publications, the lists of references of the individual publications are screened for further titles that might be of interest. This is done by applying a fixed set of inclusion rules that must be defined in advance. As this method addresses past publications from the standpoint of the one being reviewed, it is referred to as **backward snowballing**. Complementarily, f**orward snowballing** considers those publications that refer to the one being reviewed[92, p. 3]. After the whole set of literature has been reviewed this way, exclusion rules are applied to eliminate any unwanted publications. Those remaining form the basis for the next snowball sampling round. The process is repeated several times until no more new publications can be identified this way.

10.1 Research Questions

The definition of research questions (RQ) poses the first step in the process of conducting a systematic mapping study[91]. The research questions for this study should reflect the research goals: 1) the identification of relevant literature in the field of ML-SAST and 2) the extraction of the technologies being referred to. This is important as guidance for the planning of the next steps. In total, four research questions have been defined.

One important aspect in that regard is the fact that the field of ML-SAST is a conglomerate that unites three different academic disciplines: machine learning, software engineering and information security. **RQ1** aims to reflect this circumstance. So, to better comprehend the context of the publications, it is necessary to obtain an understanding of the problems discussed in the literature. In this case, the problems are the different kinds of vulnerabilities and, closely related, the programming languages addressed in the publications, see **RQ2**. In consideration of the second goal, to identify relevant technologies from the literature, **RQ3** was defined. Lastly, **RQ4** concerns itself with the question to what degree the authors evaluate their findings.

Research Question 1: To what part is each of the scientific communities mentioned involved in ML-SAST?

Research Question 2: What problems are being addressed in the literature? Which are the programming languages being targeted in the publications?

Research Question 3: What technologies and approaches are employed in the literature to solve the problems mentioned?

Research Question 4: In what regard are past findings taken into consideration? What metrics are utilized for the evaluations?

10.2 Initial Search

As previously discussed, it is necessary to define an initial set of publications as a basis for the snowball sampling process. One way to accomplish this is to **search scientific databases** for relevant literature, as suggested by Wohlin[92, p. 3]. For this, it is necessary to, first of all, define search queries (SQ) that encompass all possible research areas. This way, it is ensured that no independent publication clusters are missed[92, p. 3].

To achieve this, it was decided to formulate two search queries that put different emphasis on the research areas of **information security** and **software engineering**. A query that particularly targets machine learning was omitted since the application of ML techniques were deemed a basic prerequisite. As such, it was incorporated in both search queries. **SQ1** aims to reflect the academic discipline of information security. Preliminary searches, however, have shown that the results were often heavily diluted by publications discussing the detection of **malware**. To mitigate this circumstance, it was thus decided to exclude all literature mentioning malware using the Boolean NOT operator, which every literature database queried offered support for. **SQ2** intended to represent the research area of software engineering more closely. It could be observed that the terms **defect prediction** and **fault prediction** seemed to be deeply rooted in the software engineering community through initial test searches. As it could be argued that a vulnerability is always preceded by a fault of some sort, these terms have been included in the query. The final queries are listed below:

Search Query 1: "vulnerability" AND "learning" AND "code analysis" NOT "malware"

Search Query 2: "machine learning" AND ("defect prediction" OR "fault prediction") AND ("program analysis" OR "static analysis")

With the search queries defined, five scientific databases have been searched in total: *ACM-DL*, *DBLP*, *IEEE Xplore*, *Springer Link* and *Elsevier Science Direct*. While the first three are more discipline-specific, Springer Link and Elsevier Science Direct are more general databases. This choice was made to gain more diverse results from the searches. Moreover, the search has been limited to works published between 2014 and 2021. This was done to narrow down the search area since initial test searches brought forward no relevant publications prior to 2014. From these results, individual publications were chosen for a more detailed inspection and filtered by application of a minimal set of inclusion criteria (IN) listed below:

Inclusion Criterion 1: The publication must explicitly refer to machine learning in the context of either static application security testing or defect prediction.

Inclusion Criterion 2: The publication must be written in the English language.

Inclusion Criterion 3: The publication must have been published between 2014 and 2021.

Then, the resulting publication selection was filtered further by applying exclusion criteria (EX) and by discussion within the project team. In addition, when there was doubt about the suitability of an individual publication, experts have been surveyed for their opinion. So as to support the exclusion process, the following list of criteria was derived, depicted below:

Exclusion Criterion 1: The publication poses primary or secondary literature that discusses the application of machine learning with respect to the problem of static code analysis.

Exclusion Criterion 2: The static code analysis should be performed with the goal of detecting security vulnerabilities or, at the very least, faults that may lead to such.

Exclusion Criterion 3: The analysis should be performed based on program source code, not program binaries.

Exclusion Criterion 4: The publication must be peer-reviewed. Exemptions may apply if the publication is frequently cited in the literature.

Exclusion Criterion 5: For practical reasons, the publication must be publicly available or accessible through a publisher that the University of Bremen is subscribed to.

Literature that did not meet the requirements stated by the aforementioned exclusion criteria were excluded from the selection. The approach yielded a set of 16 publications, to which two were added by recommendation of the experts. The complete list of the literature included may be viewed in the appendix. The subsequent results for each query are shown below:

*Table 7 Iteration of all scientific database searched using queries SQ1 and SQ2, as well as the number of the corresponding results **unfiltered**, filtered by inclusion **IN** and exclusion criteria **EX**.*

SQ	***Database***	***Unfiltered***	***By IN***	***By EX***
1	ACM-DL	144	9	5
1	DBLP	3	3	1
1	IEEE Xplore	5	2	1
1	Science Direct	277	6	0
1	Scopus	3	1	0
1	Springer Link	416	9	3
1	Total	848	30	10
2	ACM-DL	83	17	6
2	Scopus	0	0	0
2	IEEE Xplore	0	0	0
2	Science Direct	70	7	0
2	Scopus	3	0	0
2	Springer Link	95	3	0
2	Total	252	27	6

10.3 Iteration Steps

As specified, the snowball sampling method is an iterative process that stretches over several rounds. Starting with an initial set of publications, each round yields another set of publications that forms the basis for the next round. The process is repeated until no more new literature can be found. As hard dependencies only exist between rounds, but not within these, the process may be parallelized. For this, at the beginning of each round, the set of publications was split across six collaborators, each of whom inspected her share for suitable literature and applied the inclusion criteria. Afterwards, the **exclusion criteria** were applied to remove any undesirable literature from the set. In addition, all collaborators came together to discuss the current's round results, stating their opinion with respect to the acceptance of each individual publication. Final decisions were reached by means of a majority vote. In total, four rounds were executed in this fashion, the number of publications yielded within each of those is depicted in the table below:

*Table 8 List depicting the individual iteration steps of the snowball sampling procedure. For each step, the resulting number of findings is reported after filtration by the inclusion **IN** and exclusion criteria **EX**.*

Round	***By IN***	***By Ex***
0	57	19
1	85	15
2	62	21
3	25	13
4	4	4

10.4 Data Extraction

Petersen et al. conducted the **data extraction** by using a **classification scheme**, which is generated with the help of a **keywording procedure**. For this step, the abstract of each publication is screened for relevant keywords that reflect the individual research contribution[91, p. 5]. Three collaborators were entrusted with this task, each sighting every publication recovered during the snowball sampling phase, at least once. Thereafter, the collaborators discussed their results and made decisions whether each respective keyword ought to be included in the scheme or not. Whenever reaching an absolute agreement was not possible, the decision was made by a majority vote. Taking the research questions taken into consideration, all resulting keywords were grouped into a total of 13 categories. Below, summaries for each of these categories may be viewed:

Asset: Whether or not the publication provides assets such as models or datasets.

Balancing: Measures taken to mitigate the effects of class imbalance. This problem occurs when a model is trained based on a dataset consisting of highly imbalanced classes. The result may be that the model would be biased in favor of the dominating class[93, pp. 167–168].

Baseline: If the authors' considered previous publications in their evaluations to draw a comparison in regard to their own results.

Community: The academic discipline the publication may be attributed to. If not directly derivable from the place of publication, e.g., venue, journal, etc., the authors' profiles and personal bibliographies were taken into consideration as well.

ANN: Artificial neural networks mentioned in the literature. These were treated separately from other machine learning techniques, as they were reported in a particularly high number and variety across the publications.

Code Partitioning: The strategies used to break up the source code into smaller chunks, suitable for processing by the machine learning models.

Embedding: The kind of embedding chosen to transform the source code chunks into a format, digestible by the machine learning models.

Classification: Whether the proposed solution performs binary or multinomial classification.

Programming Language: The programming languages targeted by the approach described in the publication.

Learning Mechanism: The type of machine learning employed as in supervised, unsupervised reinforcement learning.

Metrics: The metrics which were used to measure the performance of the approach during the evaluation of the outlined solution.

ML Technique: References to all other machine learning technologies that cannot be ascribed to the class of artificial neural networks.

Problem: The nature of the problems discussed in the literature by means of CWE-Classes, a hierarchical enumeration of security-critical software faults[85], [94]. If possible, all mentions of such classes were traced back to their base class. The aim was to prevent fragmentation as much as possible, so that comparisons could be drawn during evaluation of the results.

10.5 Summary of the Most Important Results

The snowball sampling process yielded a total of 72 highly relevant publications regarding ML-SAST. Of those. Fifty-four were classified as solution papers and eleven as evaluation papers. The remaining eight papers were considered surveys. As such surveys do not generate any new data, since they discuss the results of previous works, they were excluded from the data synthesis. However, some of the solution and evaluation papers provided assets (N=24) in the form of models (N=16) and datasets (N=16). Furthermore, a growing trend regarding the number of papers published every year could be observed, highlighting the increasing relevancy of ML-SAST. Figure 18 depicts this as a bar graph. Due to the fact that the publication dates of the inquired literature ranged between January the first 2014 and July the first 2021, the average monthly publications have been derived for each year. This allowed for the inclusion of the not yet finished year of 2021 in the trend.

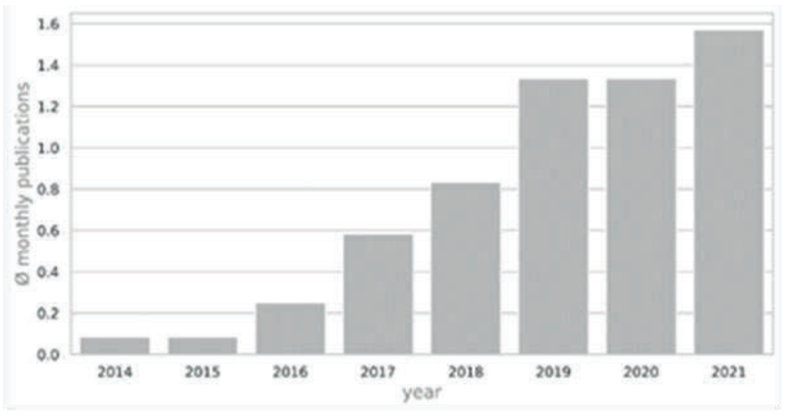


Figure 18 The normalized mean of monthly publications on ML-SAST as present in the set of sighted literature. A growing trend is visible, signifying the increasing academic interest in ML-based SAST.

Regarding **RQ1**, all the 54 solutions, as well as eleven evaluation papers, were categorized by the scientific community, the publication may most likely be attributed to. It was determined that most of the papers could either be accredited to the information security (N=42) or the software engineering community (N=37), while the machine learning community appears to represent a significantly smaller fraction (N=20). The remaining papers could not be attributed to any of the three communities. Figure 19 depicts these proportions in percentages.

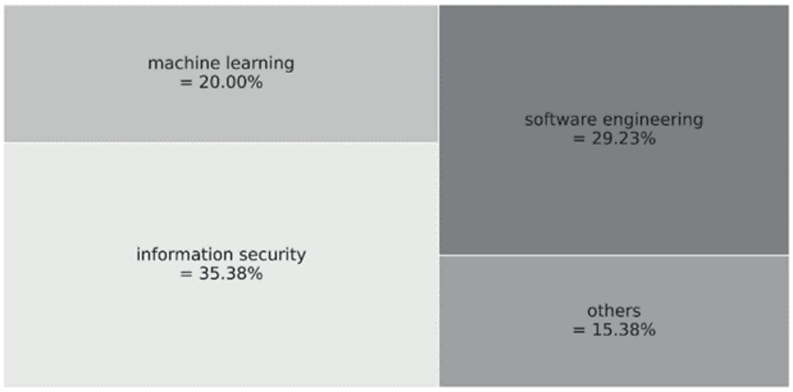


Figure 19 A high percentage of publications could be attributed to the scientific communities of information security and software engineering.

Answering **RQ2** was not as straightforward. Here two aspects needed to be considered. Firstly, the question of what programming languages are being targeted in the literature sighted needed answering. Secondly, the problems discussed in the literature must be extracted. The latter turned out to be somewhat more challenging since not all publications reverted to the usage of **CWE numbers** to describe different classes of problems. Instead, in many publications, these problems were circumscribed in an informal manner. Nonetheless, these descriptions were precise enough to be traced back to CWE classes with high confidence in most cases. Mentions of the term **buffer overflow**, for instance, could be easily accredited to the class CWE-121. Continuing with this example, all CWE classes were reduced to their so-called pillar class. For CWE-121, the parent class is CWE-787, which in turn is the child of CWE-119, which again is the child of CWE-118, a child of the final pillar class CWE-664 that encompasses all errors that underlie some sort of failure to control a resource through its lifetime[95]. A detailed description on the Common Weakness Enumeration (CWE) can be found in Chapter 7.

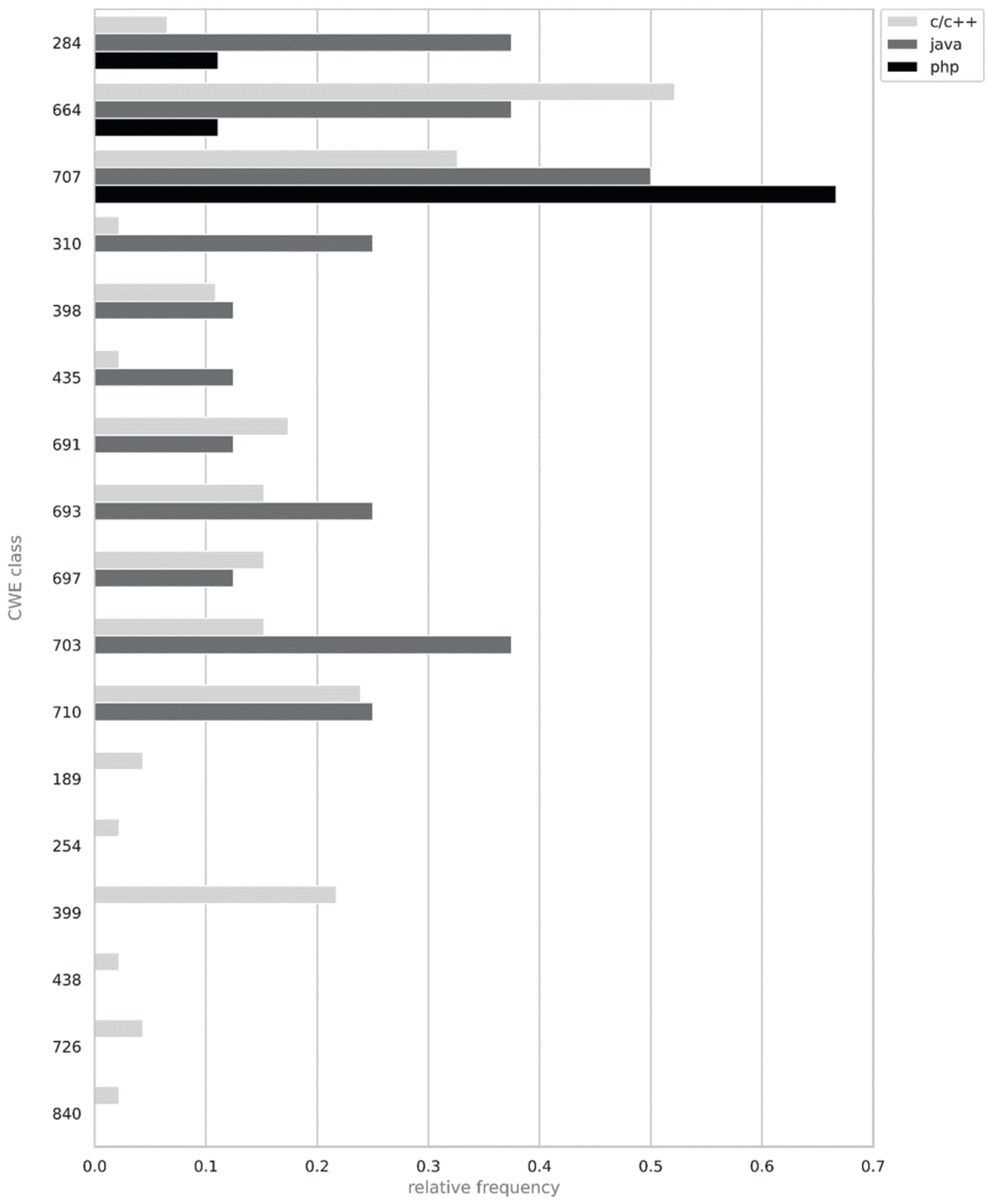


Figure 20 The number of references to the different types of software defects in the literature sighted. As can be seen, there are large fluctuations in number, depending on the programming language.

This way, all publications were sighted and CWE pillar classes extracted. It was found that the classes of failures covered in the literature appear to correlate with the programming languages targeted. In the case of publications whose approaches aimed at the programming languages C and C++ (N=42), errors that can be attributed to the class of CWE-664 were prevalent (N=24). This may not come as a surprise as both languages require the manual management of memory. CWE-707 was also often mentioned in conjunction with these programming languages (N=13). This class encompasses all errors associated with the improper validation of input[96]. Figure 20 depicts the number of times the different error classes were mentioned in the literature by mapping CWE number and programming language. The programming languages PHP (N=9) and Java (N=8) were discussed in the literature, albeit to a much lower degree. Interestingly the distribution of error classes these approaches aimed at discovering differed from that observed in C and C++. In the case of PHP, the leading class was CWE-707 (N=6), which is reasonable as PHP is a programming language predominantly used in web development. Here, the application of proper neutralization procedures may be considered crucial as adversarial user input is to be expected. For Java, again, CWE-707 was the class most often named (N=4), closely followed by CWE-703, CWE-664 and CWE-284 (N=3).

In terms of **RQ3**, multiple aspects were explored to determine which technologies the different approaches employed. First were the methods used to dissect the code into smaller segments, which are suitable for learning the models. Here, the practice of code slicing and generating code gadgets were predominant (N=22). According to the definition given by Li et al., code gadgets are semantically coherent segments of source code in terms of data or control dependence[45, p. 4]. They are thus similar to the concept of code slices introduced in Section 4.1. In some publications, the terms code gadget and code slice are used interchangeably. However, rendering a clear distinction is difficult. Another large share of publications did not rely on any elaborate strategy at all and either used function definitions as natural delimiters (N=14) or simply just split the code at arbitrary points (N=8).

Also counted was the number of times different kinds of code representations were mentioned. Tokens (N=18), abstract syntax trees (N=17), control flow graphs (N=8), program dependence graphs, and code property graphs (N=6) were most often referred to in the literature. It must be noted that these numbers merely reflect the total count each representation was mentioned in the literature. A single approach, however, may name several representations as they may be used in conjunction. Such would be the case for composite code representations, akin to code property graphs, for instance. Concerning the methods utilized to embed the features, Word2Vec was the single one most often mentioned (N=20). Most papers, however, do not mention a particular technique (N=14) or any at all (N=13).

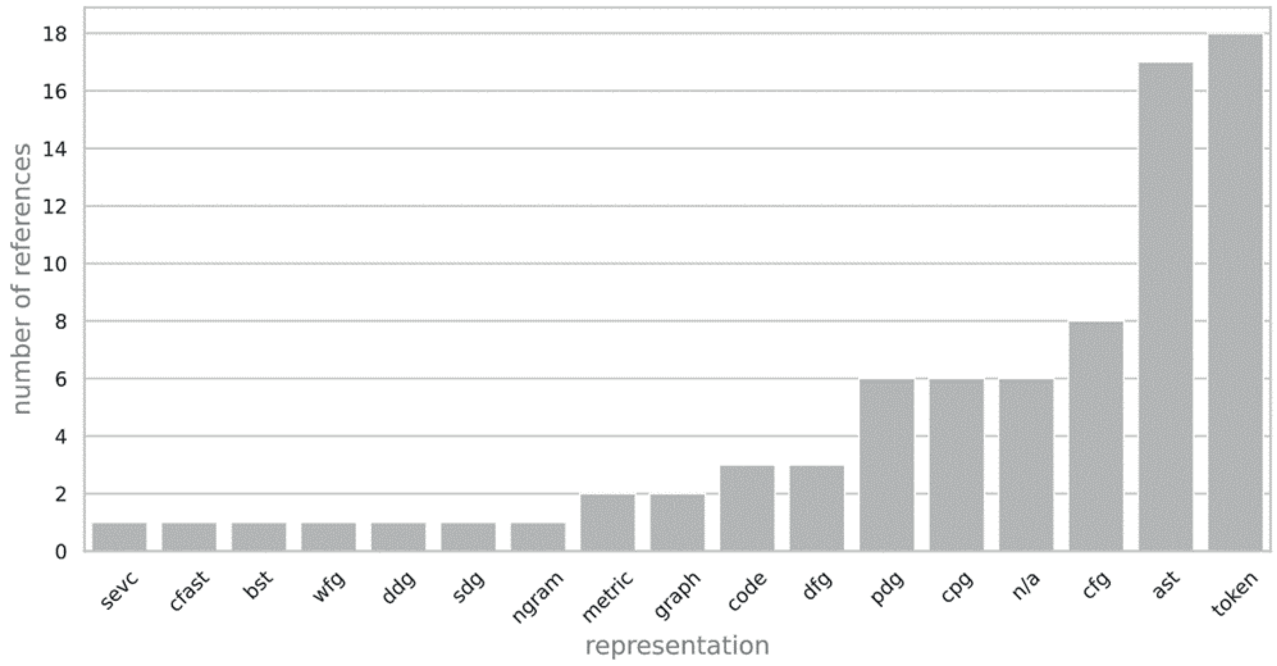


Figure 21 Regarding the different forms of code representation used in the literature, simple token or AST-based approaches are frequently chosen.

While there were publications that used more simple classification methods, such as random forest (N=10) or logistic regression (N=9), most approaches referred to some sort of artificial neural network. Within this class, bidirectional long-short-term-memory networks (BLSTM) dominated the landscape (N=19), followed by conventional long-short-term-memory networks (LSTM) (N=14). This may be due to the fact that this type of neural network, in particular, is well equipped for processing sequential data due to its contextual awareness[97, p. 8]. Unsurprisingly, the related **gated recurrent unit type (GRU)** neural networks were also mentioned in comparatively high numbers (N=6) alongside their **bidirectional counterparts (BGRU)** (N=8). Lastly, a substantial number of papers also appeared to rely on **convolutional neural networks (CNN)** (N=11) for the classification of benign and vulnerable code samples. The different types of artificial neural networks employed in the literature are covered in 2. Figure 22 depicts this visually as a bar graph.

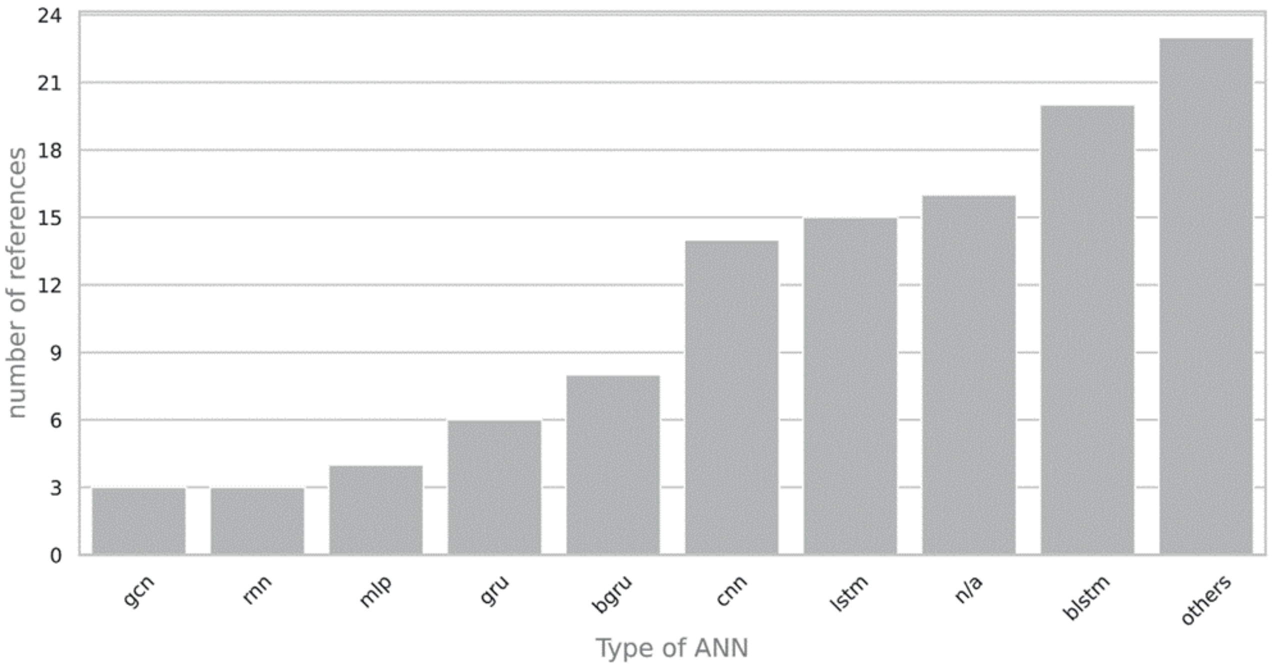


Figure 22 Concerning the types of artificial neural networks employed in the literature, bidirectional LSTMs are the most popular choice.

Finally, the datasets and associated processing steps were analyzed. As it turned out, most publications appear to incorporate real-world data from open-source software projects in one way or another (N=32). Another large number relied on more synthetic data from the **SARD project** (N=24). Strategies to counter the class imbalance within these datasets were seldomly mentioned. If so, **SMOTE** seemed to be the method of choice (N=3)[98].

To answer **RQ4**, it was found that when it comes to evaluating their own approaches, most publications used previous works to determine their own performance (N=38). Regarding the metrics used, it could be inferred that the F1-score appears to be most popular among the publications sighted (N=42). The second most one referred to was precision (N=40), followed by the false positive rate (N=35), recall (N=27), accuracy and the false negative rate (N=25), as well as the true positive rate (N=19). Surprisingly, the receiver-operator-curve (N=7) and the associated area under curve (N=9) were seldomly found within this mapping study. Figure 23 shows a bar graph of the number of times different evaluation metrics were applied. As can be seen, there seems to exist a discrepancy between the best practices discussed in Chapter 8 and the actual practices. While The F-measure, precision and recall are not the worst choices, accuracy could be considered inappropriate due to the likely high class imbalance, often present in ML-SAST related datasets, see Section 8.2 for further details.

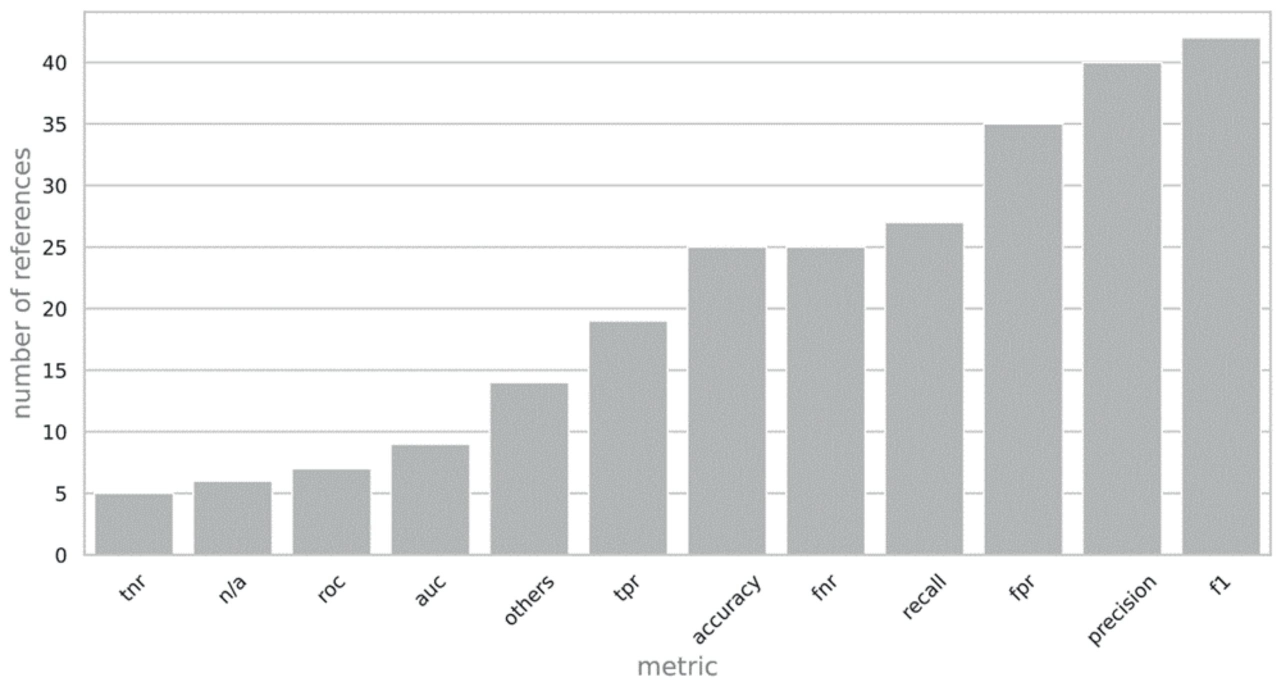


Figure 23 The number of times different metrics were mentioned in the literature sighted.

11 Detailed Description of Most Appropriate Methods for ML-SAST

The consideration of methods most appropriate for ML-SAST encompasses several key aspects, at the center of which are the machine learning algorithms. As the mapping study conducted prior seems to suggest, supervised learning appears to be prevalent in the research area of ML-SAST, see Section 10.5. For this reason, training data becomes a vital aspect that must be considered as well. Lastly, there must also be means to evaluate the trained models to determine their performance. Concerning this matter, the works by Arp et al. and Chakraborty et al. suggest that simply measuring the performance of the models in raw numbers may not be enough to satisfy this aspect. The authors found out that a deep understanding of the models at hand is crucial to assess their true performance[54], [57]. This section elaborates on the points mentioned and expounds on the most promising approaches to this end.

11.1 Training Data

With respect to the training data, several problems arise that need to be considered. First of all, a source of adequate training data must be sought out, a non-trivial task. At the time of writing, little attention has been paid to the training data in the literature regarding ML-SAST. As a result, the datasets used were oftentimes insufficient and led the models to infer false causalities by learning from artefacts present in these datasets[43]. Moreover, the data samples used are often not in line with real-world data as they are too synthetic. Models trained on such synthetic samples may perform poorly when confronted with real-world data.

Using real-world data for training purposes, however, suffers from its own demerits. Here, the challenge lies in the labeling of this previously unknown data. Doing so by the employment of human experts is most likely far too laborious to be feasible and using conventional SAST-Tools introduces a large number of samples, falsely labeled vulnerable. Some others tried to leverage data extracted from publicly available GitHub[13] repositories, inferring the labels from the commit messages, with varying success[46], [56]. In any way, the utilization of real-world data in the context of ML-SAST always seems to entail a high risk of false labeling, leading to a skewed ground truth.

Summarizing the outset, neither synthetic nor real-world data appear to be a promising option. When it comes to label noise, however, there are at least methods to mitigate the associated consequences. A decrease in classification performance is just one of such potential consequences, as stated by a survey on the topic conducted by Frenay et al. in 2014[99, p. 850]. The same survey concluded that there are different approaches to mitigate the impact of label noise: I) choosing algorithms that are more robust pertaining label noise, II) filtering out falsely labeled samples during preprocessing, and III) choosing an algorithm that actively considers label noise[99, p. 851]. Regarding algorithms that anticipate label noise, Patrini et al. came forward with a robust loss function, specifically aimed at deep neural networks. According to the evaluation conducted by the authors, their method shows a significant increase in stability when faced with noisy data in comparison to a simple cross entropy loss function as a baseline[100, pp. 6–8].

Another aspect that must be considered in the discussion of using real-world data in the domain of ML-SAST is **class imbalance**. Since the proportion of vulnerable code versus benign code samples is naturally skewed towards the latter, classifiers trained on such datasets tend to be biased towards the dominating class. Fortunately, this problem appears to be well understood, and there are known methods that offer some relief. One of such methods that have already been employed in the context of ML-SAST is the Synthetic Minority Oversampling Technique (SMOTE). Referring to this, a study by Lie et al. that has been identified during the mapping study described in Section 10.5 explored a novel fuzzy oversampling method

[13] https://github.com/

explicitly at the example of ML-SAST. According to the authors of this study, this approach showed promising results, outperforming SMOTE[101, pp. 1336–1337].

In conclusion, it was shown that a dataset comprised of real-world data appears to be better suited for ML-SAST if label noise is actively considered. Another factor that may contribute to the impaired performance of the models might be a class imbalance. Here several well-studied methods are available to combat the effects of class imbalance that should be considered for model training.

11.2 Models

As was revealed by the mapping study, RNN type neural networks seem to be the most popular choice for ML-SAST in the literature. This circumstance might be explained by the fact that these kinds of neural networks are well suited for processing sequential data, such as source code[97, p. 8]. The problem with this approach is, however that while the raw source code indeed is a flat sequence of tokens, semantic interdependencies may better be expressed through graph-like data structures. In the case of recurrent type neural networks, these data structures must first be flattened again before being fed to the model. This may possibly lead to a loss of information[54, p. 11], [102, p. 729]. As far as more conventional kinds of neural networks are concerned, recurrent type neural networks appear to perform slightly better when compared to convolutional neural networks as several evaluation studies suggest[22, p. 31], [97, p. 8].

One such evaluation study, conducted by Zheng et al., also compared neural networks' performance against more conventional machine learning algorithms. The authors compared **convolutional neural networks**, **BLSTM networks** and **gated recurrent unit networks** against **random forests**, **k-nearest neighbor**, **support vector machines**, **gradient boosting decision trees** and **simple logistic regression**. They found that neural networks generally perform better, measuring an average increase of 18.78% in F1-score, 8.41% in precision and 24.71% in recall[22, pp. 30–33].

There are examples, however, that employ an entirely different kind of neural network, called **graph neural networks (GNN)**. This type of network does not require the input data to be converted into a low-dimensional vector space to be processed. Instead, the network actively considers the information provided by the graph structure itself[102, p. 1]. Graph neural networks operate on the principle of neighborhood aggregation. The idea was originally proposed in a paper by Gori et al. In 2005.

Graph neural networks act by assignment of labels and states to the graph's nodes and edges. These states are defined by a transition function fw that considers the label of the node itself as well as the states and labels of its direct neighbors. Moreover, each node's output may be computed by solving an additional output function gw. Through **Banach's fixed point theorem**, it is guaranteed that if fw is designed in a way that the composition of all of its instances Fw presents a contraction mapping, then there exists a unique solution to that function with respect to the state of the graph. Hence, fw and gw form a parametric function pw(G, n) that given a graph G and one of its nodes n computes the output of that node. The underlying machine learning task lies in the iterative adjustment of the weights w that affect fw and gw so that the functions approximate a given set of training samples. Gori et al. achieved this by minimization of a quadratic loss function.

Furthermore, Banach's theorem also states that if Fw is a contraction mapping, then it will converge exponentially fast towards a fix point for any initial state. Graph neural networks solve this by replacement of the graph's nodes with units that calculate fw. On activation, these units will compute the next state of the node based on its previous state. When a stable fix point has been reached, the weights are adjusted by a gradient descent strategy in accordance to that point. This is repeated until an arbitrary stopping criterion has been reached[102, pp. 730–731]. This way, with each iteration step, a unit learns more about its surrounding neighbors, influencing its state. It must be noted, however, that this description is a heavily simplified summary and, in addition, based on the earliest paper by Gori et al., which has later been refined[103]. Based on this simple version of GNNs, a plethora of derivative works exist that either improve on some of its aspects or adapt the model to some other type of neural network, such as graph convolutional neural networks. Zhou et al. have compiled a comprehensive survey, including a vast list of different GNN architectures[104].

Cheng et al.[55] as well as Zhou et al.[56] state to have successfully deployed GNNs for the detection of vulnerabilities, reporting a significantly increased F1-score in comparison to a 3-layer BLSTM network and a CNN[56, p. 8].

The approach by Zhou et al. relies on a gated recurrent unit type graph neural network (GGNN) and **code property graphs** to detect vulnerable code on a function level granularity[56, p. 3]. The authors report trouble, however, extracting the CPGs from some of the function's source code. To facilitate a more efficient graph embedding, the authors use edges denoting last read and last write relations instead of the full data flow graph[56, p. 6]. Moreover, the natural sequence of code is preserved by providing additional edges between adjacent tokens of the AST[56, p. 5].

The output of the GGNN is then run through a novel convolutional layer that comprises multiple one-dimensional **convolutional layers** and **maxpooling layers**[56, p. 5]. A model was trained using manually annotated functions from four different open-source software projects. Each dataset was randomly shuffled and then split, using 75% of the samples for training and the remaining 25% for evaluation purposes. The authors evaluated their approach against three traditional SAST-Solutions as well as a simple **CNN** and a three-layered **BLSTM** with and without an attention mechanism. On average, the authors report a relative increase of 10.51% in accuracy and a relative gain of 8.68% concerning the F1-score across all datasets when compared to the baseline methods.

Alas, the authors fail to report the distribution of vulnerable and benign samples in the dataset. Without this information, it is unclear how dependable the numbers of the reported accuracy are. Moreover, the authors used samples from the same software projects for their evaluation. Therefore, it is unclear how robust their results truly are. The model could, for instance, have overfitted, thus performing well on the present datasets only[56, p. 5].

The other more recent approach proposed by Cheng et al. is more complex than that by Zhou et al., as it also considers interprocedural vulnerabilities. For this, first, the PDG is generated from the interprocedural control flow graph and value flow graph (also known as data flow graph or DFG, see Section 3.2.2) of the given source code. Then, system API calls and arithmetic operators are chosen as so-called *points of interest*. Originating from these, subgraphs of the PDG called XFGs are extracted through forward and backward slicing[55, pp. 7–8]. Like the approach proposed by Zhou et al., the natural code token sequences are used as features as well. Cheng et al. generate **Doc2Vec** embeddings from the normalized source code of each XFG extracted[55, pp. 8–10]. As slicing and the normalization of code can lead to sample duplications, the authors took measures to remove such samples prior to training[54], [55, p. 10].

The authors evaluated three different types of graph neural networks: Graph convolutional networks (GCN), graph attention networks (GAT) and higher-order graph neural networks (k-GNN)[55, p. 5]. Each graph convolutional layer is followed by a pooling layer. Based on the output of all pooling layers, a global average and maxpooling operation is performed. The results are then aggregated before being fed to a multi-layered perceptron for the final task of binary classification[55, pp. 10–13]. Model training was performed based on the *SARD* dataset, as well as two real-world open-source software projects: *Lua* and *Redis*. The latter datasets were manually labeled in the same manner that Zhou et al. already employed. The authors consider an XFG vulnerable if it contains at least one statement of the code that was annotated as vulnerable.

For the evaluation, the authors extracted a total of 409,262 benign XFGs as well as 156,195 vulnerable XFGs from the SARD dataset. In addition, 2,848 benign and 1,323 vulnerable XFGs were extracted from the real world open-source software projects[55, pp. 13–16]. The training was performed using 10-fold cross-validation, based on randomly selected samples from 80% of the combined dataset. The remaining 20% were used for the detection phase[55, pp. 17–18]. According to the authors, k-GNNs performed best on average, with superior performance in accuracy, false positive rate, false negative rate, true positive rate, precision, F1 score, AUC, informedness and **markedness**. Only in terms of runtime, the k-GNNs fell short, taking the last place when compared to the GCNs and GATs[55, pp. 18–20].

The approach was then evaluated against four conventional SAST tools: Flawfinder, RATS, Clang Static Analyzer and Infer, using the reserved 20% of the samples extracted from the SARD dataset, outperforming each by a large margin[55, pp. 20–21]. When compared to three other ML-based SAST approaches: Token-based, VGDetector and VulDeePecker, according to the authors, their approach again outperformed all others across all metrics[55, p. 38].

Taking a closer look at the architecture proposed by Gori et al., it quickly becomes evident that one of its downsides is the expensive mechanism by which the states are propagated across the network. This severely limits the scalability of GNNs, making them unsuitable for large program graphs. To overcome this problem, Wang et al. proposed further improvements to GNNs called graph interval neural networks (GINN). These GINNS are specifically designed for the purpose of handling graph representations of programs. Combined with their novel code representation of hierarchical intervals, based on control flow graphs, the authors claim that this solution exhibits a higher precision even across large networks[105, pp. 2–3].

In summary, graph neural networks seem to be a promising technology with respect to ML-SAST. They appear to be better suited to learn from the rich semantics that graph-like program representations offer when compared to other types of neural networks.

11.3 Explainability

According to the evaluation studies conducted by Arp et al. and Chakraborty et al., the inference of false causalities due to the models picking up on irrelevant features during training appears to be an underestimated problem in the literature[43, pp. 8–9], [54, p. 11]. In such cases, the models learned to recognize patterns that loosely correlated with the problem at hand, a circumstance that might not be recognized by a purely quantitative evaluation of the model using some select metrics. Ideally, measures to prevent the model from learning such false causalities should be taken in advance. However, even if the utmost care is taken, it is impossible to tell for certain if the model has inferred false causalities. Several techniques exist that aim at mitigating these problems by finding explanations for the model’s behavior. More precisely, given a specific input, these methods try to determine to what degree the individual features of that input influenced the model's prediction[57, p. 3].

A tangible example of this may be found in the domain of computer vision. Here, it is imaginable that a simple convolutional neural network could be trained to classify images by their content. Shown below in Figure 24 is an image of a speckled horse that was fed to a VGG16 classifier, trained on a subset of the **COCO train2017 dataset**[106], [107]. Interestingly the model misclassified the image as showing a dalmatian. To a practitioner trying to understand what led to this misclassification, methods such as **layer-wise relevance propagation (LRP)** can help gather a high-level understanding of the model and its underlying mechanics. The method can generate a heatmap that can be laid over the image, highlighting the most relevant pixels for the prediction the model made. The output has been generated using a simple implementation of the LRP algorithm, provided by Montavon, a contributor to the original LRP paper in 2015[108], [109]. As can be seen, the model appears to have put most of its attention on the lower speckles and the legs, highlighted in red. The outline marked in blue, on the other hand, had a negative impact on the predicted class. Although finding definitive answers as to why the model has mistaken the horse for a dalmatian is not possible, it could be argued that the speckles presumably led the model to its prediction. With that knowledge, attempts could be made, for instance, to adjust the training dataset in a way so that future predictions will be more accurate[109, p. 194].

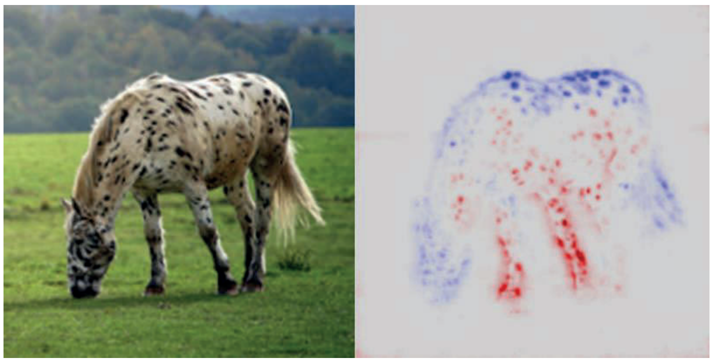


Figure 24 Shown on the left is the image of a horse, that was erroneously classified as portraying a dalmatian by a VGG16 network. The image on the right depicts the heatmap obtained by applying the LRP method to the model and input.

Given a prediction f(x) for the input x in $\mathbb{R}^x$, Layer-wise relevance propagation works on the principle of relevance scores $\mathbb{R}_d$ that are propagated through the network in opposite direction. For this, the contribution of each individual node in a layer l to the activation of each node in the follow-up layer $l + 1$ is computed, relative to that of all other nodes in l [108, p. 194]. Applied to the whole network, LRP is a decomposition of the prediction f(x) into a sum of terms, so that the equation below holds true. Here $\mathbb{R}_d$ is defined over the input x of dimension V [109, pp. 2–3].

$$f(x) \approx \sum_{d=1}^{v} R_d$$

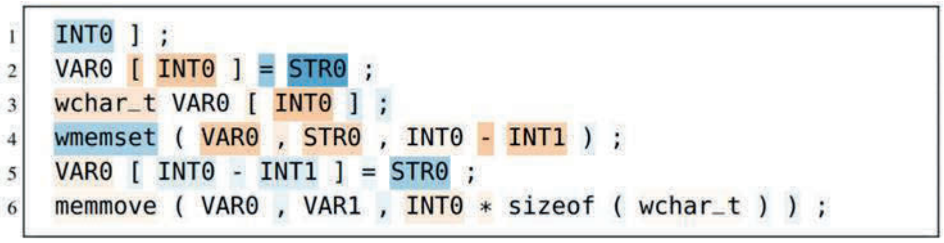
The process terminates at the network's input layer, where in the case of computer vision, the final relevance scores may be mapped to a corresponding color space[108, p. 195], [109, p. 46].

The same problem can be easily applied to ML-SAST as well. Here, the images are replaced by code samples and instead of wondering what pixels have had the highest relevance with respect to f(x), the question, relies on the tokens most relevant. The same way a heatmap can be applied to a given input image, it is then possible to highlight the tokens according to their relevance score. This is possible as the LRP method has been adopted to a variety of different neural network architectures, such as **multilayer perceptron networks**, **convolutional neural networks** as well as **LSTM networks**, commonly employed in the context of ML-SAST[57, p. 3].

Warnecke et al. were the first to explore the application of methods such as LRP in the context of information security[57, p. 1]. The authors conducted a conclusive evaluation, analyzing six different explanation methods concerning their eligibility in the context of security. Generally, there are two distinguishable types of techniques: **Black-box** and **white-box methods**, i.e., those that do not assume knowledge about the model and its parameters and those that do[57, p. 3]. Regarding the more **general evaluation criteria**, the authors analyzed the methods' **descriptive accuracy** and **descriptive sparsity**. In basic terms, the methods ability to highlight the relevant and only the relevant code tokens for a prediction[57, pp. 8–9].

In terms of the more security-related metrics, **completeness**, **stability**, **efficiency** and **robustness** were measured. A method that is complete allows computing meaningful relevance scores for all inputs x and its according prediction f(x). Moreover, a stable explanation method always delivers the same results for each run on the same inputs. The efficiency refers to the computational complexity of the method, and finally, the robustness describes the methods sensitivity to adversarial perturbations[57, pp. 9–11].

Concerning the methods analyzed, the authors chose three **black-box methods** for analysis: **LIME**, **LEMNA**, **and SHAP**. Additionally, the authors also evaluated three **white-box methods**: **gradients**, **integrated gradients (IG)** and **LRP**. The results were that white-box methods are generally far superior to black-box techniques. Of the latter, however, LIME offered the best performance. When it comes to white-box methods, LRP showed the most promising results, although the overall difference between those methods appears to be rather minute. The evaluation has shown that LRP offers the second-best accuracy and sparsity while being complete, efficient, and stable across different runs. None of the methods tested was robust against adversarial perturbations. However, a property most likely neglectable in the context of ML-SAST as no adversarial input is to be expected[57, p. 11], [97]. Shown below, in Figure 25, is a code segment that has been used as input for a classification model. Using integrated gradients, a heatmap has been generated to visualize which of the input features influenced the prediction. Warm colors present a strong influence on the predicted class. Cold colors present features that excerpted a negative impact for the predicted class[57, p. 6].



```
INT0 ] ;
VAR0 [ INT0 ] = STR0 ;
wchar_t VAR0 [ INT0 ] ;
wmemset ( VAR0 , STR0 , INT0 - INT1 ) ;
VAR0 [ INT0 - INT1 ] = STR0 ;
memmove ( VAR0 , VAR1 , INT0 * sizeof ( wchar_t ) ) ;
```

Figure 25 A code segment highlighted using integrated gradients. Orange colored tokens express a high contribution to this segment being predicted vulnerable. Blue colored tokens contradict this classification.

Unfortunately, it is not possible to apply these methods in a straightforward fashion to Graph type neural networks, as their architecture is fundamentally different from that of the networks previously described. The main difference lies in the fact that not just the node's labels carry meaning but also the way in which nodes and edges interconnect, i.e., the graph's topology. In fact, such a graph could have completely unlabeled nodes and edges and still present a meaningful data structure[74, p. 2]. When looking for explanations to a specific prediction f(x) for an input x, this kind of information should also be considered as well, hence why methods such as layer-wise relevance propagation fail.

Despite not having received the same level of attention as more conventional types of neural networks regarding explanation methods, in the recent past, several such techniques have been introduced for GNNs as well. Contrary to the conventional architectures, however, there does not appear to be a catch-all solution for graph neural networks such as is LRP for the conventional architectures. This, at least, is what Yuan et al. found when they conducted a taxonomic survey exploring different types of explanation methods for GNNs. Furthermore, the authors argue that the choice of technique is highly dependent on the number of variables.

There are two categories of techniques: Instance level explanation methods and model level explanation methods. Instance level methods deliver model explanations for a specific input, while the latter provides input-independent explanations. Whereas instance-level explanation methods seem to be well explored, model level methods appear to have only just emerged. In fact, the survey only analyzed one of such techniques, named XGNN. Concerning instance-level explanation methods, there are four general concepts that need to be differentiated[74, p. 3].

Gradient methods use backpropagation to calculate the gradient of a given prediction f(x) and its corresponding input x. Likewise, the related feature-based techniques rely on the interpolation of hidden features onto the input space.

Perturbation methods work for a given prediction f(x) by continuously masking out features of the corresponding input x, resulting in the altered input x'. In theory, if important features are being retained, then no significant changes between f(x) and the prediction based on the altered input f(x') should be observable[74, p. 4].

Surrogate methods try to approximate the original model by means of a simpler model that can be explained more easily. With graph neural networks, such approximations are not possible, as the topological information is of discrete nature. In other words, an edge either connects two nodes or it does not, but there exist no possible in-between states. Nevertheless, attempts have been made to explain GNNs using surrogate models[74, p. 6].

Decomposition methods, to which the previously introduced LRP method can be attributed, work by decomposing the model into a sum of terms. Here each term represents the relevance score of a specific input dimension. Fulfilling this conservational property is not trivial in the case of GNNs, as these scores would have to be distributed with respect to the graph's topology[74, pp. 7–8].

These methods all come with their own merits and deficiencies. While gradient-based methods that Yuan et al. explored are simple, they do not account for the information contained the graphs topology. Instead, they merely consider the labels of the edges and nodes for the task. The perturbation-based methods described by Yuan et al. do consider such information but are mostly involve learning procedures. On the one hand, methods based on learning appear to be better at capturing the relationships between the predictions and their inputs, according to Yuan et al. On the other hand, these may be less trustworthy if such procedures involve another black box that needs explaining. In this regard, all the surrogate methods described by Yuan et al. rely on learning that involves a black box. The decomposition-based methods do not rely on learning, but in most cases, do not account for the topology specific information encapsulated by GNNs. Only GNN-LRP does so, however, at the cost of computational complexity. Lastly, the one and only instance level method described by Yuan et al. called XGNN also involves black box-based learning but poses the only technique capable of delivering general explanations for a model[74, pp. 8–9].

The survey also included a framework to evaluate the different methods by means of fidelity, stability, sparsity, and accuracy. While the latter metrics are as previously described, fidelity measures the methods ability to identify input features that are important to the model.

Finally, it must be noted that Ganz et al. have conducted a comprehensive evaluation, published towards the end of this study, which explores different explanation methods for GNN-based SAST approaches. Consequently, it could not be taken into consideration at the time of writing. The authors evaluated a variety of graph-agnostic methods, such as **LRP** and **integrated gradients**. As for the GNN-specific methods GNNExplainer, PGExplainer and Graph-LRP were assessed, based on their **descriptive accuracy**, **structural robustness**, **constrastivity** as well as **graph sparsity**. The evaluation was carried out based on three different GNN-based SAST approaches from other publications, comprising Devign, ReVeal and BGNN4VD[54], [56], [110]. The findings of the authors appear to be in line with the results from other publications, already discussed. Ganz et al. report a significantly better performance of the graph-agnostic methods in terms of descriptive accuracy, robustness, stability and efficiency, whereas the graph-specific approaches only achieve a vastly better sparsity and moderately better contrastivity, when compared[111]. These findings suggest that unlike is the case with more conventional architectures, e.g., convolutional or recurrent neural networks, there appears to be no catch-all solution concerning suitable explanation methods for graph neural networks. In respect thereof, Ganz et al. state:

> *“We find that all explanation methods have shortcomings in at least two criteria and therefore hope to foster research for new explanation methods.”[111, pp. 11–12].*

In addition, the authors remark that many explanation methods seem to focus on unimportant features, but achieve a large descriptive accuracy, which leads the authors to believe that the models have picked up on spurious correlations and in fact did not learn to identify vulnerabilities[111, p. 10].

12 Prioritization of Models for this Approach and Outlook on Further Work

As the previous sections have highlighted, there are multiple options regarding the models suitable for ML-SAST. Each of these models come with their own merits and demerits, rendering the conception of a strictly ordered prioritization a challenging task. In addition to this, such prioritization is highly dependent on the research goals aimed at. Yet another difficulty arises from the circumstance that the comparison of the different models presented in the literature is a non-trivial task, as there is no common baseline between these. In addition to that, poorly chosen metrics may belie the fact that some individual models exhibit insufficient generalizability [54, p. 10], [57, pp. 3–12]. Aside from that, it must also be noted that the predictive performance of any model is not merely the result of the architecture itself, but also highly influenced by the data it has been trained on. Albeit not unimportant, the choice of architecture should therefore be regarded to as one of many puzzle pieces in the overall picture. Equally, if not more important, are also the quality and the preprocessing of the training data.

This chapter, however, is not meant to discuss the latter of such matters. Instead, suitable options in terms of machine learning algorithms for ML-SAST shall be determined in the following sections. Five different possibilities for future research on ML-SAST were identified, which will be elaborated on in this section, highlighting their merits as well as demerits regarding different research goals. Finally, a conclusion is drawn as to which approaches should be prioritized and which direction appears to be promising regarding further research.

12.1 Graph Neural Networks

Graph neural networks, introduced in Section 11.2, are a rather new approach in machine learning, which shows promising results across a variety of research disciplines. If the goal is strictly defined to be outperforming other approaches in some metrics, then graph neural networks could be argued to be the contender most suitable for this task. In comparison to other architectures, this type of network may offer the benefit of considering the data, comprised in the topology of the numerous graph-like representations of source code, to a higher degree[112, p. 13]. According to some studies on ML-SAST that incorporate graph neural networks in their solutions, this results in an overall better performance in contrast to other architectures, such as LSTM networks and conventional SAST approaches[54, p. 14], [55, p. 25], [56, p. 8].

Due to the fact that the availability of literature on graph neural networks for SAST is sparse, the results in this area of research should be considered with some distrust. It could, however, be argued that this sparsity leaves room for further improvements, not present in more traditional ML-SAST approaches, where only incremental advancements are likely. Of the few studies identified to this end that rely on graph neural networks, the architectures used include: recurrent-, gated- and convolutional graph neural networks[54]–[56], [113]. Wu et al. have generated a comprehensive survey on a variety of graph neural network architectures. One of such architectures, which appears to have been previously ignored in the literature, are so-called spatial-temporal networks, capable of inferring knowledge from spatial as well as temporal correlations[114]. With respect to the success of recurrent neural networks for ML-SAST, which work on the basis of a similar premise, this architecture may be worthwhile investigating further.

Yet another downside of graph neural networks is their lack of explainability. To this end, there are no known methods capable of delivering the same degree of model explanations for graph neural networks, as is the case for recurrent neural networks[74, p. 1]. As shown in two independent evaluation studies by Arp et al. and Chakraborty et al., being able to deduce some explanations for the model's behavior is critical, as the lack thereof may lead to significant deviations between the expected and the actual behavior of the models[43, p. 3], [54, p. 2].

In conclusion, graph neural networks appear to offer some benefits over other models in regards to ML-SAST, albeit at the cost of explainability. As an emerging field of research, it is possible that this architecture still offers room for more significant advances than other architectures, like recurrent neural networks. However, choosing this approach over a more established one poses a certain risk as the research has not been very advanced yet and given the small body of literature available, it is difficult to assert the true potential of graph neural networks for ML-SAST with definitive certainty.

12.2 Conventional Neural Networks

The literature mapping described in Chapter 10 revealed that there is a strong bias towards recurrent neural networks for ML-SAST. This may not be very surprising as source code may be viewed as a sequence of tokens so that such sequence models are an evident choice. Moreover, a lot of approaches seem to rely on **preprocessing** steps that generate various kinds of tree- or graph-like structures from the source code in combination with some sort of **slicing mechanism** to train the models on. In this study, these types of approaches are referred to as conventional strategies for ML-SAST due to their high frequency in the literature.

As some evaluation studies have pointed out, there are several areas in which more traditional approaches, which rely on recurrent neural networks and some form of composite code representation, could be improved. One of the more difficult challenges in ML-SAST, however, is still the **lack of sufficient training data**. Chakraborty et al. reported drastic performance increases when retraining existing models from other literature on their own dataset[54, p. 9]. Moreover, Liu et al. introduced a novel fuzzy **oversampling** technique for class rebalancing. The authors tested their approach against other **rebalancing methods** for ML-SAST, reporting improvements across several metrics[101, p. 1337]. Another aspect of improving on might also be the active consideration of label noise. Real-world data, i.e., code segments derived from actual software projects that are annotated as either vulnerable or benign, is oftentimes plagued by a high level of mislabeled samples[43, p. 3]. Patrini et al. have conducted a comprehensive evaluation on factors helpful in alleviating some of the negative effects of noisy data[100]. Regarding the code samples, another aspect worthwhile considering is their form of representation. The evaluations conducted by Zhen et al. as well as Chakraborty et al. seem to suggest that the Semantic Vulnerability Candidates, introduced by Li et al., are a capable form of representation[52], [54, p. 9], [115, p. 652]. Other factors that may show an impact on performance include the choice of the **deep learning algorithm**, the **overall architecture of the model** and other **hyper parameters**.

It must be noted that this enumeration of aspects is inchoate and does not reflect the situation to its full extent. It is merely meant to exemplify the areas in which an improvement on the more established approaches may still be feasible. In summary, despite the comparatively long time in the context of ML-SAST, in which research on conventional approaches using recurrent neural networks has been conducted, there may still be room for incremental improvements in some selected areas.

12.3 Clustering Approaches

Clustering presents another approach to ML-SAST that appears to have received less attention than binary classification. Instead of learning a classification model based on a set of labeled data samples used to recognize the patterns it has been trained on, clustering identifies groups of data that show similarity. This approach may be considered interesting for the reason that when asked how machine learning could be employed for SAST, three out of seven experts voiced unsupervised learning as a viable option during the interviews described in Chapter 5. One of the experts with a strong background in AI, went on to describe a method not unlike the one presented by Ahmadi et al. The interviewee stated that it might be possible to use a clustering algorithm to first group the data, before a model could be learned in a supervised fashion, based on these clusters

Considering these opinions, gathered during the interview, the question arises as to why clustering-based approaches are underrepresented in comparison to the ones relying on binary classification. It could be argued that due to the high influence that the VulDeePecker approach has excerpted on the community, other possibilities may not have been taken into consideration. It may also be the case that given the task of telling vulnerable from benign code binary classification is the much more obvious approach to solving this problem.

Whilst the use of recurrent neural networks for ML-SAST appears to be the most popular choice, according to the data gathered throughout the literature mapping in Chapter 10, there are, aside from graph neural networks, other very different and noteworthy approaches. Despite being potentially underrepresented in the previous sections of this study, these approaches are not necessarily inferior in comparison to other approaches. One approach that stands out from the rest is called FICS and was proposed by Ahmadi et al. in 2021[65]. Unlike the other approaches that commonly rely on model-based machine learning techniques, i.e., techniques involving a pre-trained model, Ahmadi et al. use an instance-based clustering method that solely relies on the code base to be analyzed. For this purpose, the authors first extract intraprocedural data dependency graphs. Then a two-step clustering approach is employed that first attempts to find similar constructs within the graphs extracted prior. The clustering algorithm used is DBSCAN, which does not require the number of classes to be known beforehand[65, p. 9]. In the second clustering step, graph embeddings are constructed within the clusters, using the graph2vec algorithm and the distances between the embeddings are measured. Variances in these distances may be indicative of a vulnerability and thus reported to the user, who may then verify the validity of these warnings. The authors report plausible, albeit moderate, improvements over conventional SAST solutions, including APISan, LRSan and Crix. Specifically in terms of the rate of true positive findings, a noticeable increase in performance can be seen[65, p. 12].

One of the more apparent drawbacks of this approach is the fact that it only works intraprocedurally, unlike some of the competing solutions. There may also be chances for improvement through the usage of other, possibly more suitable, clustering algorithms. This makes the approach of Ahmadi et al. an interesting contender for further research.

Another, albeit early, ML-SAST clustering approach stems from Yamaguchi et al.[42]. It builds upon CPGs[36] and uses interprocedural static analysis to extract code artefacts similar to slices. Then it attempts to cluster sources (e.g., attacker-controlled entry points) with respect to given sinks (e.g., parameters of memcpy or malloc). The clustering is based on the so-called Jaro distance metric that is particularly well suited for short strings[42]. Distance metrics are introduced in Chapter 2. The approach also tries to automatically infer sanitization rules for the considered sinks, i.e., rules that are required for the safe usage of a sink's arguments. As a result of this clustering task, the technique allows one to infer graph traversals, which in essence are queries on the CPG. On applying such queries, taint-style vulnerabilities can be detected as demonstrated by several use cases (including the Heartbleed vulnerability)[42]. Although it is an earlier ML-SAST approach, employing clustering seems to be still promising and could be implemented, for example, based on the analysis infrastructure made available by the Joern tool. As rule definition is a common, but often tedious task in the context of SAST tools, rule inference appears to be helpful when using SAST tools and finally may relieve security analysts from the burden of rule definition. Furthermore, the inferred rules are largely comprehensible, which makes them valuable for security analysts (cf. the explainability problems related to techniques based on neural networks). In addition, the results of the expert interview support to also consider clustering-based approaches (see Chapter 5).

Lastly, Fischer et al. proposed an approach that employs clustering to learn pattern embeddings that encode similarity information[116, p. 346]. The authors' motivation stems from the notion that developers often reuse code samples from Stack Overflow[14], a platform where users discuss programming-related questions. This platform may be considered unreliable, as it does not guarantee the correctness in terms of the code samples provided there as answers. Reusing such insecure samples in production code can lead to serious vulnerabilities. However, Fischer et al. also found that while there are insecure samples, in 99.37% of the cases the same platform also hosts the secure implementation of the same sample[116, p. 339]. The paper describes a ML-based solution that provides users of the Stack Overflow platform with information

[14] https://stackoverflow.com/

about the correctness of the code samples and in case they are shown an insecure implementation, nudges them towards a post that provides a correct implementation.

The difficulty lies in the fact that in many cases the code samples provided on the platform are incomplete and cannot be compiled with a regular compiler. Instead, partial compiling methods must be employed, introducing ambiguities, such that samples despite depicting the same usage-pattern, lead to different PDGs. The resulting features are referred to as unsound, or respectively, sound in case they are completed by the authors. In order to alleviate this problem, the authors learn similar pattern embeddings that are independent from the surrounding program by employing a so-called Siamese neural network. This network is capable of learning code pattern representations that preserve the similarity properties. For this task, the network is trained using pairs of sound and unsound samples of the same code in an unsupervised fashion. These pairs are generated from open-source projects found on GitHub[15], first generating a set of sound methods with a regular compiler, then based on the same methods, generating the unsound set through partial compiling. Subsequently, transfer-learning is applied to train the network to distinguish between secure and unsecure samples and their usage-pattern, in a supervised fashion[116, p. 346], using manually annotated samples from Stack Overflow.

12.4 Reinforcement Learning

As can be seen, supervised learning appears to be the leading technology in ML-SAST. The reasons as to why that might be the case can only be conjectured about. Given the fact, however, that Ahmadi et al. were able to demonstrate the feasibility of unsupervised approaches to the problem, the question arises, whether **reinforcement learning** could also be used for this purpose. This idea has also been proposed by one of the interviewees in Chapter 5. Given the problems associated with the lack of sufficient training data, as described in Section 11.1, this approach could pose an interesting option to overcome these challenges. It could be conceivable that such an approach could be realized as a service that hosts a central model. Developers could use this service to examine their source code for vulnerabilities. Once the model has made its predictions, the developers would then review these and could themselves label the predictions as either correct or false, generating new training data for the model. This way, their expert knowledge could be leveraged to continuously improve the model, which in turn would benefit the developers.

Despite this approach appearing to be good in theory, there are certain obstacles that need to be taken into consideration. First of all, and rather evidently, there would have to exist some central model that could be accessed by all developers. This would require some platform to host the model, which generates running costs. Secondly, there are security-related issues that need to be addressed as well. As this central model processes the source code of all developers, their information will be persisted in the model in some mostly unpredictable manner. Under certain circumstances, it may then be possible for an adversarial user with malicious intents to extract this data from the model, gaining confidential information[117, p. 308]. Chen et al. proposed an attack on deep reinforcement learning models that, according to him, is capable of stealing models with great accuracy and fidelity[117, p. 318]. However, not only would the confidentiality of the data be at risk, an adversarial user could also deliberately falsely annotated training samples, so as to coerce the model into ignoring certain vulnerability patterns. The latter of these problems may be mitigated by some peer-reviewing process, which on the other hand, would make the service cumbersome to use and could lead to a drastically decreased acceptance by developers. Another solution could be to measure the distance between the training samples provided by the developers. Then, only the samples provided by different users that lie in close proximity to each other and share the same label could be used to train the model. It is, however, unclear if this approach would be feasible, as it may require a high number of users and vulnerability patterns, occurring rather seldomly, could possibly be ignored.

[15] https://github.com/

12.5 Ensembles of Approaches

One last option that ought to be discussed is the combination of multiple approaches discussed so far. Instead of attempting to improve on a single solution, there may be benefits in leveraging the capabilities of several approaches at once. Drawing comparisons between the different approaches proposed in the literature is a non-trivial task, as their characteristics can drastically differ. However, it may be a possible to turn this detriment into an advantage if some of the rather dissimilar approaches are combined into an ensemble.

For example, Ahmadi et al. and Yamaguchi et al., both described the applicability of clustering algorithms for ML-based SAST. One of the key strengths of these approaches is the fact, that they learn from the software that is being analyzed. In contrast to the supervised models that are learned on external program samples, unsupervised models may thus be much better at detecting intrinsic defects, such as API misuse. However, the same mechanism might as well be considered their biggest drawback. If the software that is being analyzed simply does not contain enough samples of some program behavior (or any of such samples at all), the approaches described by Ahmadi et al. and Yamaguchi et al. would likely not be able to pick up on them [42, p. 809], [65, p. 2037]. Vice versa, any supervised method may not be very good at detecting the aforementioned kinds of intrinsic defects, as they would not have been trained on them.

This example highlights the potential that ensemble solutions might hold. As these approaches are likely to detect very different vulnerability patterns, it would be difficult to prioritize one over the other. Their aptness would be highly dependent on the task at hand. Whereas the unsupervised approaches appear to be highly adaptable and therefore capable of detecting more intrinsic defects, supervised techniques may be better at discovering more general types of defects. By combining both solutions, it may be possible to leverage the benefits of both while simultaneously compensating for each of their demerits. Such endeavor would require the careful selection of some of the approaches identified during the literature mapping in Chapter 10, based on their unique strengths and weaknesses. Using a standardized benchmarking framework that would need to be conceived prior, different ensembles could be systematically compared and the most suitable elected.

12.6 Conclusion of the Prioritization

In summary, there is no best approach to be chosen for further research. Which of the various solutions discussed in this literature study is most capable, remains a question highly dependent on the task it is intended for and most of all the requirements imposed.

Graph neural networks show promising results in raw detection performance, as their ability to consider the information contained in the various graph-like forms of program representation appears to be highly beneficial for the purpose of ML-SAST[54, pp. 9–13], [55, p. 25], [56, p. 8]. This advantage, however, comes at its own demerit, as the methods for finding explanations for the predictions made by graph neural networks are still not on par with those of the more conventional neural network architectures, such as recurrent or convolutional networks, see Section 11.3 [74, p. 13]. Moreover, as the idea of employing graph neural networks for the purpose of ML-SAST is a rather novel one, there is a lack of literature available on that topic. Consequently, it is difficult to reason about the validity of such few results, as there is no strong consensus regarding the aptness of employing graph neural networks for ML-SAST. It must thus be noted that further research in this direction also entails an increased risk of yielding underwhelming results.

Another direction of attempting to improve on some of the more conventional approaches, which rely on some recurrent type of neural network, comes with the benefit of being thoroughly documented in the literature. It is thus possible to resort to a large body of readily available knowledge, including some valuable evaluation studies which investigated the influence of various factors on the performance of different ML-SAST approaches as well as common pitfalls[21]–[23], [43], [54], [97], [114], [115], [118]. Moreover, the research on explanation methods for the more traditional architectures of neural networks is very advanced, allowing for precise per-prediction explanations. It is, however, necessary to highlight the circumstance that any possible improvement on these approaches would likely be of incremental nature.

Lastly, going entirely different routes might also pose an option. In view of the approaches by Ahmadi et al. and Yamaguchi et al., there may lie possibilities in the continuation of their ideas[42], [65]. Both publications offer some very concrete connecting factors for further research in the discussion of their respective limitations. Additionally, and possibly most promising, it may also be possible to achieve further improvements through combing several approaches into an ensemble. This way, for instance, the high adaptability and the capability of unsupervised approaches to detect more intrinsic defects, could be combined with supervised approaches, which may be better at detecting the more general types of software defects.

As a final conclusion on which method to prioritize, it could be argued that the expert interviews in Chapter 5, as well as the literature identified, showed that explainability is an important requirement for SAST. As such, it would be a logical conclusion to dissuade from using graph neural networks to this end. Thus, the following options remain:

1. Conventional ML-SAST approaches that rely on recurrent neural networks may be improved on, see Section 12.2.
2. Clustering approaches, similar to those of Ahmadi et al. and Yamaguchi et al., could be continued, as they show potential, but have been largely neglected in the ML-SAST community[42], [65].
3. Combinations of any of these methods, i.e., ensembles of different approaches with different strengths.

13 Prototypical Implementation of a ML-SAST Tool

As an extension, the insights gained throughout the literature study ultimately culminated in the creation of an ML-based SAST tool. It is planned by the BSI to publish the tool under the MIT license, such that it can be obtained by anyone interested. This chapter comprises a preliminary discussion of the underlying problems of applying machine learning to SAST, the ideas behind this prototype, and its conception. Although some mitigations to the aforementioned difficulties could be achieved, it may be preempted that the application of machine learning techniques to the problem of SAST, remains challenging. It appears as though there is a gap between the issues described in the literature and the actual issues faced in reality. The following sections cover these challenges in detail and the solutions worked out. Furthermore, the development process of the prototype and its evaluation will be described as well.

13.1 Presuppositions and underlying problems

Starting at the very foundation of ML-based SAST, as has been discussed in chapters 12 and 14, several categories of machine learning algorithms may be used to this end: Supervised, unsupervised and reinforcement learning. In these chapters, one of the key points established, was that supervised ML-SAST approaches pose a challenging endeavor due to their requirement of labeled training data. As of writing this document, existing datasets of this sort could be considered inadequate, as they tend to be either too artificial or suffer from great amounts of label noise. Section 11.1 elaborates on problems associated with training data and in extension, supervised ML-SAST approaches in general. Furthermore, due to a lack of literature on reinforcement learning based SAST techniques, this branch of machine learning was not explored during the development of the prototype, hence it will not be discussed here. Lastly, clustering methods were also considered during the development of the prototype and in fact, several options were explored to this end. This was due to the aforementioned problems, associated with supervised methods that do not apply to clustering methods, as they do not require labeled training data. Unfortunately, they too suffer from issues that will be discussed in the following sections. Aside from this, more general problems that do not pertain to either approach in particular, will be discussed here as well. Due to the pitfalls that are associated with supervised methods and in particular their requirement of labeled training data, the problems associated with clustering methods will be discussed here first.

13.1.1 Unsupervised Methods

Unsupervised methods may come with the benefit that they do not require such training datasets, since they merely partition the data into clusters. In the case of ML-SAST, this may be, for instance, the separation of source code segments into benign and vulnerable clusters. As has been elucidated in the literature study, it was found that the vast majority of ML-SAST approaches made use of supervised learning and not clustering algorithms, raising the question as to why the latter appears to be such an unpopular choice within the community. It is impossible to pinpoint the exact reason for why this might be, but despite the numerous shortcomings of supervised approaches, it may be the case that interpreting the results of clustering algorithms poses a difficult problem. Based on the observations made during the development of the prototype, the following may be considered a factor in this context as well.

The basic working principle of clustering methods is, that they partition data on the premise of observable patterns, as long as they occur in sufficient frequency. This alone is not helpful, however, when trying to tell vulnerable code apart from benign code. Therefore, it is necessary to further process the results of the clustering algorithms, for instance, by applying additional heuristics. One of such heuristics may be broken down to the assumption that anything that is observed many times must be benign and important. Applied to a program’s behavioral patterns, one can see how this approach could be used to model wanted behavior within a program: If some behavior can be observed repeatedly it is likely correct and also likely to be important. Both, the approach by Ahmadi et al. and Yamaguchi et al. rely on this idea to some degree [42], [65]. Attempts at using this technique during the experimentation phase were to no avail, unfortunately, as it was impossible to yield satisfactory results. It appears that the patterns that were extracted from various graph program representations could not be observed in adequate number to reliably deduce rules. It must be added that this problem may also be attributed at least partially to the particular types of graphs used or a lack of suitable embedding methods for graphs in general. The following sections will shed more light on these issues in particular. Especially for the embedding methods, however, it is unclear how far the latter issue is solvable.

Regardless of these challenges, on a more general note, it may be added that unsupervised approaches showcase a tendency to shift workload from model training to the interpretation of the results. Furthermore, unsupervised approaches may only be able to reason based on what is there, due to the fact that they use the software project to be analyzed to generate the model. This has the benefit, that API specific defects could be potentially detected with greater success but dangerous patterns where benign counterparts are not available in sufficient frequency or available at all will not be picked up on by the tool. This is not a problem, but rather a property of unsupervised approaches that practitioners should be aware of and that should be made clear to the final users of ML-based SAST tools. Supervised methods, on the other hand, may fail to recognize the intrinsic problems that are specific to some software as they would have to be explicitly trained with samples representative of these problems. This way, the types of bugs each of the types of ML-based SAST approaches may recognize, could be considered to be mostly disjoint sets. As for the prototyped described in the following chapters, it is attempted to strike some balance between both worlds, by using a supervised approach that relies on highly adaptable models and a feedback loop. This solution offers the benefit of being able to be trained on arbitrary samples from other software projects, but also being able to be quickly adapted to intrinsic problems, based on prior findings. The prototype will be explained in detail in Chapter 13.2.

13.1.2 Embeddings

As indicated in the previous section 13.1.1, embeddings play a vital role in machine learning and therefore also in the domain of ML-based SAST. It is difficult to find a unanimous definition of the term in the context of machine learning. Informally speaking, embedding methods aim to map some input data into a possibly low dimensional vector space, while preserving the structure of the input data to the largest degree possible. That is, at least for the information that is important for the learning algorithms.

Except for, maybe, graph neural networks, the embedding of program segments is a challenging task that appears to be largely overlooked in the literature. As the quantitative evaluation of the literature study has shown in Section 10.5, many of the ML-SAST approaches appear to rely on the embedding of simple code tokens. In its most simple form, this idea is a very natural approach to the problem at hand, i.e., generating real-valued vectors from the source code. Since the code already has the structure of a one-dimensional vector of tokens, it must only be mapped onto some numerical representation. To this end, a plethora of techniques, such as the well-known word2vec method, are readily available. This practice suffers from two major drawbacks, unfortunately. First, since the models may easily pick up on delimiters and other special characters, they are likely to learn coincidental correlations [43]. This can be corrected to some degree. Secondly, this form of program representation does not adequately reflect on the intricate dependencies that are found within a program. As a result, the underlying defects may not be appropriately represented by the embedded code segment.

Alternatively, it is possible to use well-established static analysis techniques, to first generate graphs from the source code that are able to represent these different relationships within a program. This approach is much more robust, since statements that may not be adjacent in the code, but are still connected by data or control flow, may be effectively mapped into a vector space in close proximity to each other. Some better-known examples of these forms of program representation are the Program Dependence Graph (PDG) or the System Dependence Graph (SDG) [37], [119]. Section 3.2.2 elaborates on these data structures in more detail. It may seem that this technique is undoubtably the better choice, but unfortunately there are also drawbacks associated with this method. In contrast to the much more naive token-based approaches, here the input data for the embedding methods are not already present in the form of one-dimensional lists of tokens. Instead, these program representations are graphs, which are much harder to map onto real-valued vectors. Except for graph neural networks, introduced in Chapter 12.1, all other machine learning techniques, whether they are supervised or unsupervised, require this form of input. Graph neural networks could be therefore considered better suited in that regard. Unfortunately, they are also entirely opaque and their interpretation remains a challenging task [74], [111].

While there are methods for the embedding of graphs, they are scarce and suffer from some severe issues. Firstly, it appears that a lot of the methods either attempt to embed the graphs topology or the data associated with each node, but rarely both. For program representation graphs, both types of data are important, as will be explained later in this section. One exception that does take the topology, as well as the data contained within the nodes into account, is the Graph2Vec method. During the development of the prototype, attempts were made to employ this method for ML-SAST. To this end, the Bourne Again Shell (BASH) in version 5.1, was compiled into the LLVM intermediate code representation (LLVM-IR). Then, using the Static Value Flow (SVF) tool, the interprocedural control flow graph (ICFG) was built for the BASH. SVF is a powerful framework for static code analysis tasks. It comprises numerous advanced points-to analysis algorithms and is capable of generating a variety of graph program representations. Finally, for each procedure in the program, the subgraph was extracted and embedded using the aforementioned Graph2Vec method. It was hoped, that when clustering the embedded functions, they would form separable clusters in the vector space. Alas, this was not the case and even structurally similar functions would frequently fall into entirely different clusters. Using the T-Distributed Stochastic Neighbor Embedding (t-SNE) dimensionality reduction method was applied to plot the thus embedded procedures, depicted in Figure 26. Each datapoint shown represents a single procedure.

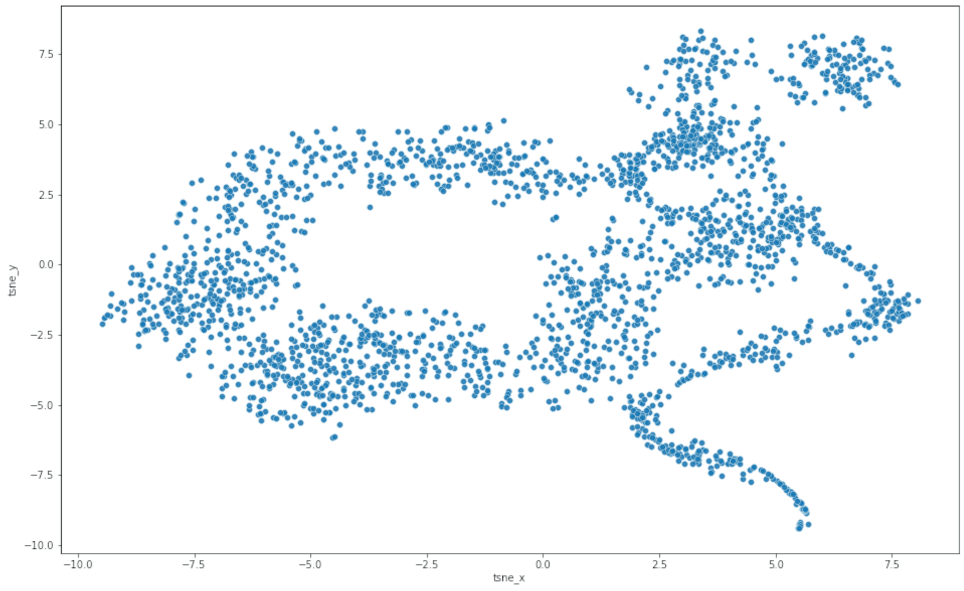


Figure 26: ICFGs of all procedures of the Bourne Again Shell, embedded using the Graph2Vec method and plotted using the t-SNE dimensionality reduction technique.

As can be seen in this plot, the different procedures are hardly separable. Despite the fact that different parametrization of the t-SNE dimensionality reduction method may lead to drastically different results, using the DB-SCAN as well as an agglomerative clustering algorithm, the apprehension that the Graph2Vec method is possibly unsuitable for this usage scenario was only strengthened. In another attempt, the LLVM-IR of the BASH was used to first generate graphs by linking dependent instructions over their operands and results. This graph was, again, embedded using the Graph2Vec method, but this time the used feature, was the OP-code associated with each instruction. The resulting t-SNE plot drew a very similar picture: The individual function being hardly separable in clusters, see Figure 27. It is unclear what exactly caused this poor performance. It may be the case that the extraction of entire functions was simply too large, drowning any potential patterns in noise. This choice, however, was motivated by the publication of Ahmadi et al. who reported success, embedding entire procedures in a similar fashion [65]. Furthermore, the extraction of entire functions is very amenable, since the information of which instruction belongs to which procedure is readily available and does not require complex traversals of the graph. The extraction of all paths between some source and some sink for taint style analyses, would have been very complex, for instance. Usually, taint-style analyses only attempt to answer whether there is any single path between two points in the graph, which, for example, can be accomplished using Dijkstra's shortest path algorithm. For a clustering approach, however, this would not suffice, as it is necessary to extract all paths between two points in a graph to find common patterns within them. This problem is computationally very hard and the process to do so, simply took too long to be feasible in the experiments.

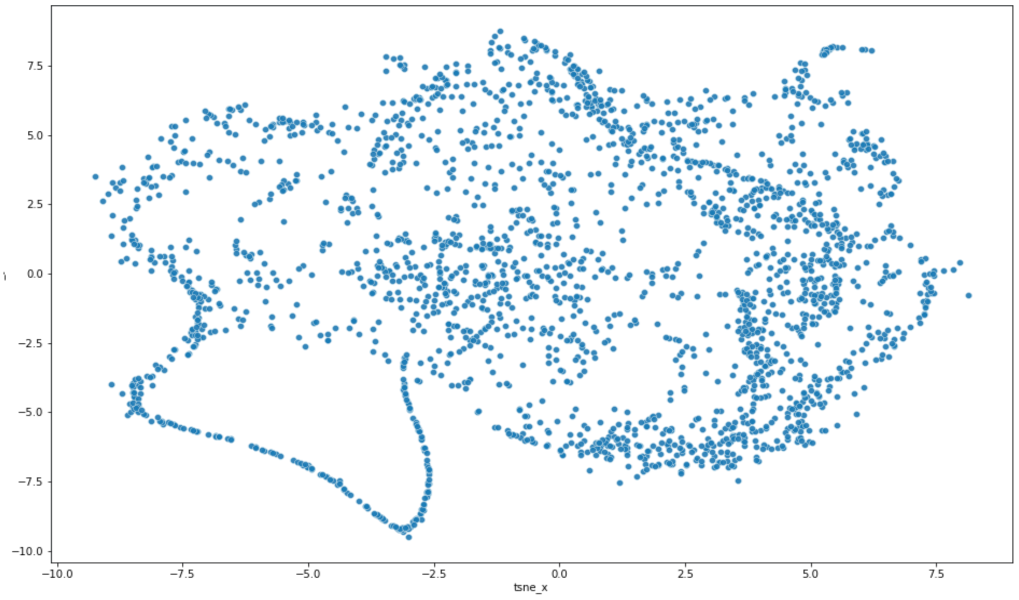


Figure 27: Each procedure of the BASH, embedded again, but this time using the LLVM-IR OP-codes as features, embedded using Graph2Vec and plotted using the t-SNE method.

These examples highlight the fact that embedding complex graphs into a metric vector space is not an easy task. As mentioned earlier, the number of appropriate methods to this end is small and may be considered one of the most challenging problems in the context of ML-SAST. Such embedding methods are very important, due to the fact that most of the available machine learning techniques require these graphs to be mapped into a metric space first. This is necessary in order to make distances measurable, see Section 2.4. For comprehensiveness' sake, it must be mentioned that graph neural networks may be considered in exception in this sense. This type of neural network is capable of learning from the graph's topology directly, but at the price of not being explainable at all, see Section 11.3. For program representation graphs, this task is especially hard. Not only must the embedding method pay respect to the topology of the graphs, the graphs are often of heterogeneous nature, which means that their nodes and edges have different types. The different types correspond to different semantics. For instance, when considering the ICFG alone, there are nodes to express procedure calls, procedure returns, points in the program where the control flow diverges or merges and simple instructions, i.e., instructions that do not directly affect the control flow. This information must not be neglected, as it is very valuable in the detection of software defects. In addition, these nodes also may hold even more information that may be beneficial in this regard. A call or return node, for example, may be annotated with the name of the calling procedure or, vice versa, a function call node with the name of the callee. Other, simple instruction nodes may also hold the type of instruction as additional information. Figure 28 depicts such an ICFG, with the corresponding C-code listed in Figure 29.

It must be noted that the graph shown does not directly correspond to the source code. This is because the graph was generated using the SVF-tool, which is based on the LLVM compiler infrastructure project, hence using the LLVM intermediate code representation. The SVF-tool will be explained in greater detail in Sections 13.1.3 and 13.2.2. As can be seen, even a simple piece of code may lead to a comparatively large

ICFG. A large contributing factor to this is of course the fact that the graph was generated using the LLVM-IR and not the source code directly. There are few alternatives with respect to the analysis software available that is capable of generating such program representation graphs. This issue will be further elaborated on in Section 13.1.3. Not only are such graphs large and complex, but they also often contain additional information in their nodes and edges, such as the type of node.

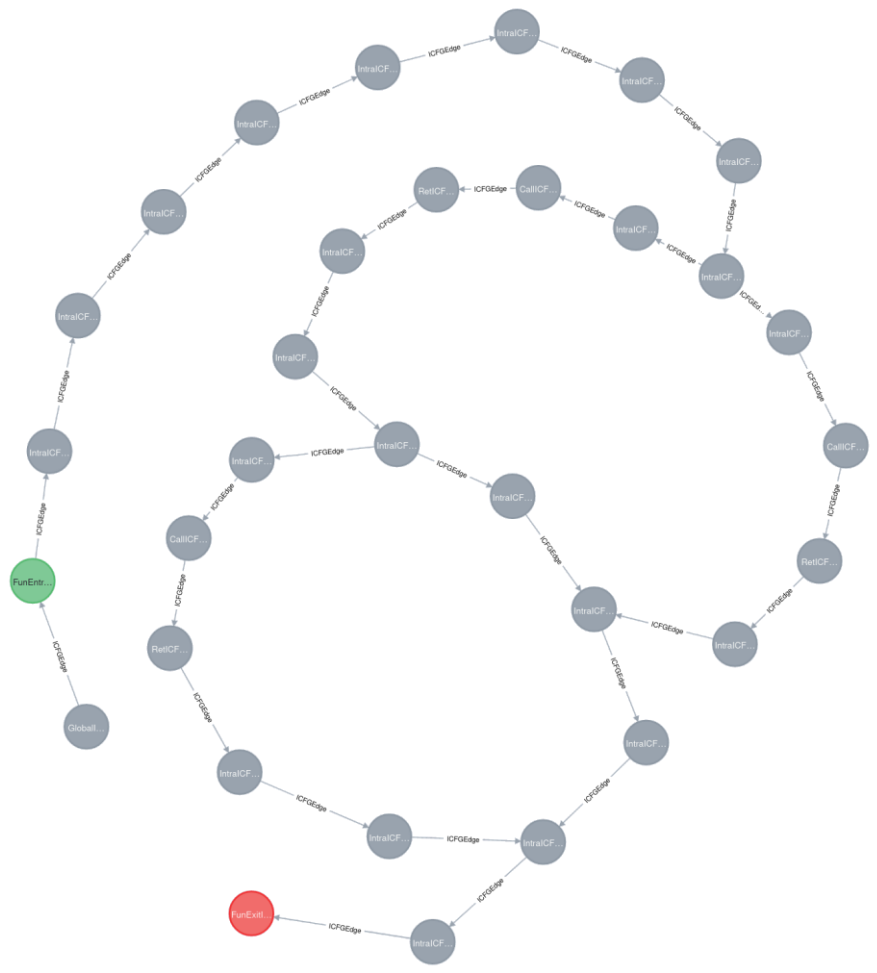


Figure 28: The ICFG of a very simple program, listed below in Figure 29. Note that the ICFG is generated from the LLVM bit code of the C-code.

```c
int main(int argc, char const *argv[])
{
    if (argc < 2) {
        printf("argc < 2");
    } else {
        printf("argc >= 2");
        if (argc > 3) {
            printf("argc > 3");
            return 1;
        }
    }
    return 0;
}
```

Figure 29: The C-code that corresponds to the ICFG above in Figure 28.

As can be seen, program representation graphs often hold an abundance of data that should not be neglected. At the same time, this data, including the graphs' topology itself, must be mapped into a vector space with very limited dimensionality. Instead of using more complex embedding methods, another way to mitigate these challenges is to model the problems to be analyzed in such a way that the graph traversals necessary yield very simple data structures instead of complex graphs. This can be achieved when limiting oneself to paths only, where neither the incoming, nor the outgoing degree of each node exceeds one. Then, these paths may be mapped into a metric space by, e.g., assigning each of the different types of nodes a numeric value and constructing a vector, where each dimension corresponds to a single node in order of traversal.

On a more theoretical level, the problem of embedding graphs into a metric vector space may be attributed to the specific properties of graphs. It is likely that the difficulties observed when embedding graphs are the result of the graph isomorphism problem, i.e., whether two graphs are the same. This problem is believed to belong to the class of NP-intermediate problems [120]. It is therefore unlikely that there is an amenable way of assigning graphs a canonical representation in a metric space, as this would essentially solve the graph isomorphism problem. Hence, only approximations to this problem can exist. Some work by disassembling the graph into graphlets first before generating hashes over these graphlets [121].

In summary, it is hard to pinpoint the exact reasons for as to why the results yielded using the Graph2Vec algorithm were so underwhelming. It is not the intent to say that the Graph2Vec method, and possibly other graph embeddings, are generally unfit. There are many factors that could have led to these results. The algorithm could have possibly not been parametrized correctly, or the extraction of the entire procedures could have been a bad choice for the subgraphs to be embedded. Furthermore, it must also be made clear that although plenty of attempts were made to use the Graph2Vec embedding method, no systematic experiments were conducted, which could strengthen these assumptions. This was due to the fact that the idea of embedding complex subgraphs was abandoned early on, hence rendering the application of graph embedding methods unnecessary.

On a different note, the embedding of graph program representations may be a particularly challenging subfield of the already challenging field of graph embeddings. This may be due to the possibly repetitive nature of these graphs, as they are always a very condensed form of the program at hand and the patterns therein may be too similar in many cases. This may lead to closely adjacent points in the space. Furthermore, differences between a benign and a vulnerable procedure often only ever impact a handful of nodes or edges in these graphs, making them virtually indistinguishable in their embedded form. This is unsurprising as these graph representations were never created with machine learning applications in mind, leaving this task up for further research. Graphs offer many benefits for conventional static analysis approaches, in particular the possibility to express many problems as simple reachability problems. For the particular use case of ML-SAST, however, it may be argued that other data structures may be better suited.

13.1.3 Lack of Software

Another unforeseen obstacle was the lack of static analysis tools that are being developed under an open-source license. As already elaborated in Section 13.1.2, the use of well-established static analysis methods precursory to the actual application of machine learning techniques may offer some benefits. Firstly, a Points-To-Analysis (PTA) is necessary, in order to estimate the memory objects a pointer variable may point to at any time during program execution. This information is not only important in terms of data flow, but also for the control flow of a program, since function pointers that are often being used in call back patterns, could otherwise not be resolved. This, in turn, would mean that every consecutive application of machine learning algorithms would rely on imprecise data, possibly altering its predictive performance negatively. Aside from a PTA, the generation of program representations that reflect program particular semantics, such as the control flow or the data flow, may also be helpful in increasing the conciseness of the input of the machine learning algorithms, as mentioned earlier.

While a multitude of commercial solutions for this exact task exist, free and open-source implementations are scarce, at least for the C and C++ programming languages. During the development of the prototype, three different tools have been evaluated: Joern, DG and SVF. Beginning with Joern, the tool uses Code Property Graphs (CPGs) at its core, which are essentially an amalgamation of the CFG and the PDG, which are connected over the program's abstract syntax tree (AST) [36]. When tested, the Joern tool stood out positively by offering a comprehensive interface for querying the graph and for its speed. Another point where the tool stood out positively, is its capability to parse incomplete program code. This reason allows for a much larger number of datasets to be used, where the complete source code is not available, but merely smaller excerpts. On the downside, it must be noted that the tool does not employ a points-to analysis, which rendered the tool unsuitable for the development of the prototype. It must be remarked in this context that when asked about the presence of a points-to analysis, one of the authors mentioned this feature being slowly introduced at the moment[16]. Once Joern supports this feature, however, attempts could be made to incorporate the tool into the prototype.

Another tool that was closely inspected was the DG-Tool. Originally developed as part of a master's thesis, the tool is meant to create program slices of LLVM-IR code, to be used as part of the Symbiotic verification tool [122]. To this end, DG does perform a simple Andersen's PTA and generates so-called Dependence Graphs (DGs). These graphs are an incomplete form of the well-established SDGs. This circumstance, the fact that the tool is not able to process C++ code and finally stability issues led to the abandonment of DG after some testing.

Finally, the SVF tool was examined for its suitability. It is also based on LLVM-IR code and offers a number of very precise and fast PTA algorithms to be chosen from and has the ability to generate several types of program representations, such as the Call Graph with points-to information (PTACG), the Interprocedural Control Flow Graph (ICFG) and lastly the Value Flow Graphs (VFGs) or Sparse Value Flow Graphs (SVFGs). The graphs depict the flow of values throughout the program, i.e., it mostly contains the relationships between statements that define some variable. Moreover, SVFGs only consider those statements that are either directly or indirectly reachable from a memory allocation site, making them very concise [123]. This has benefits, but also drawbacks, as this conciseness makes SVFGs arguably less intuitive than SDGs that adhere more closely to the original syntax of the program itself. Another contributing factor to this may be the fact that SVF also works on the basis of LLVM-IR code. This, although being very reasonable from a developer's standpoint, adds another layer of abstraction. Finally, unlike the other tools mentioned, SVF is published under the GPLv3 license, which may be problematic for developers who want to incorporate the tool into their software without having to publish their source code.

Finally, another suitable tool that unfortunately could not be tested due to a lack of awareness thereof, is Phasar[17]. Similarly to SVF, Phasar is built around the LLVM-Compiler Infrastructure and like the former allows for complex dataflow problems to be solved. Unlike SVF, the tool is licensed under the MIT license, which may be beneficial for some developers as mentioned before [124]. Again, it may be possible to incorporate the tool into the prototype in the future.

As can be seen, the choices are limited in terms of static analysis software that comprises the needed functionality. All options that could be made out came with some drawback, limiting their use. Although the algorithms necessary to create program slices from SDGs have been thoroughly described by Horwitz et al. since the early nineties, it appears that their implementation is very difficult [37]. This trait may also be traced all the way back to the high complexity of the C and C++ programming languages themselves. Based on these considerations, it was decided that the best suited tool for the implementation of the prototype would be the SVF framework, see Section 13.2.3. As the SVF framework was the most mature solution, the limitation of being published under the GPLv3 license was deemed to be an acceptable drawback.

13.1.4 Summary

In summary, it was revealed that despite the downsides of supervised machine learning algorithms in the context of ML-based SAST, the application of clustering methods to this end, appears to remain challenging. Moreover, the embedding of arbitrary subgraphs is a challenging task. At least under circumstances present, it was not possible to find a suitable configuration that would yield sufficiently separable embeddings in the vector space. This problem may affect program representation more than graphs present in other domains. This may be due to the fact that these kinds of graphs tend to be repetitive but

[16] https://github.com/joernio/joern/issues/1128
[17] https://github.com/secure-software-engineering/phasar

also very complex at the same time. Software defects also often only impact very few nodes, complicating matters even further. The development of suitable forms of graph embeddings for SAST may be considered a key element to allow for advancements to be made. This task, however, is up for further research.

Moreover, finding appropriate analysis tools that would allow for the generation of such program representation graphs was equally not without difficulties. Free, open-source tools that combine a sound PTA with adequate program representations and slicing capabilities are rare for the C and C++ programming languages. To this end, it appears that the best solution currently available is the SVF tool [125]. In contrast to the former problem, this is may not be a basic research question, as the necessary algorithms are known. It would still require extensive resources to develop such a tool, which may explain why there are so few available.

Finally, as has been explained in the previous Chapter 11.1 of the literature study, appropriate training data appears to be virtually non-existent at the time of writing. The datasets available are either too synthetic or suffer from label noise, which in turn yields models that generalize very poorly.

13.2 Development of the Prototype

As has been elaborated, there are three challenging factors that complicate the development of a machine learning based SAST tool: A lack of suitable graph embedding methods, a lack of appropriate training data and finally a lack of sound, free and open-source analysis tools. The latter of these problems could be addressed to some satisfactory extent by the usage of the SVF tool, hence this problem will not be discussed any further in this section.

13.2.1 Basic Approach

In Section 11.1 the lack of suitable training datasets was already discussed in detail. In order to make use of supervised machine learning models, however, the availability of these is indispensable. In summary, training datasets may fall into one of two categories: Synthetic datasets, i.e., those that are purposefully crafted by hand and datasets from real vulnerabilities extracted from various software projects. Whereas the models that were trained on synthetic datasets appear to generalize poorly due to the simplistic nature of the samples therein, the models trained on real-world data seem to generalize poorly as well [43], [54]. The possible reasons for this are more complex and possible causes include label noise and class imbalance. Another factor may also be a lack of context as these datasets tend to only extract the individual procedure that is affected by the vulnerability. Even for human experts it may then be impossible to determine whether the sample shown is vulnerable or not, because crucial information is missing. The ReVeal dataset, introduced by Chakraborty et al., is a real dataset that contains defective functions and their repaired counterparts from various software projects [81]. The example shown in Figure 30 was taken from this dataset and highlights the aforementioned problem. The procedure was taken from the QEMU project and contains a serious flaw. Figure 31 depicts the same procedure, but this time after fixes have been applied. Without additional information, however, it is almost impossible to tell the samples apart, since the defect appears to be very specific to the APIs of QEMU.

```
static void cpu_request_exit(void* opaque, intirq, intlevel) {
    CPUMIPSState* env = cpu_single_env;
    if (env && level){
        cpu_exit(CPU(mips_env_get_cpu(env)));
    }
}
```

Figure 30: A vulnerable sample taken from the ReVeal Dataset.

Tracing the changes between both samples back to commit that introduced the fix, another problem becomes visible: Label noise. As the authors explain, they have used the Debian security tracker to extract the samples, likely in an automated fashion. This way it appears that the samples shown here are not representative of an actual defect, but instead are part of some larger code refactoring effort [18]. Therefore, the Code in Figure 30, may not be vulnerable to begin with, thus posing a prime example for the kind of label noise one might find in realistic datasets.

```
static void cpu_request_exit(void* opaque, intirq, intlevel) {
    CPUState* cpu = current_cpu;
    if (cpu && level) {
        cpu_exit(cpu);
    }
}
```

Figure 31: The fixed sample of the same procedure depicted in Figure 30.

In defense of the ReVeal dataset it must be noted, however, that this is just one example and as such does not allow for a comprehensive judgement to be made about the quality of this dataset. Doing so was never the intended purpose of this example, as it merely serves for the elucidation of these otherwise abstract problems. A systematic evaluation of the ReVeal dataset against the background of the aforementioned issues, would be a matter for future research.

It may also be the case that the samples extracted from real software projects alone, are too few and too heterogeneous. Simply training a supervised machine learning model on samples extracted from real-world software leans heavily on the idea that among all of these very different samples few, unified and discernible patterns exist that allow for the discrimination between benign and vulnerable code. While disproving the existence of such patterns is likely as difficult, if not impossible, as it is to prove their existence, evaluations of existing ML-SAST models may provide some insights. Firstly, when confronted with other datasets than the original test datasets, it appears that the models perform drastically worse across all metrics [81]. Thus, if there were such general patterns, the datasets either failed to contain these patterns or the methods applied must have failed. As said, these are vague cogitations for now and further research is necessary to uncover the intricate relationships between the training data and the generalizability of machine learning models in the context of ML-based SAST.

As a consequence from this observation, one might be tempted to opt for unsupervised machine learning methods instead. At first glance, this seems like a good alternative as this method does not require labeled training data. Some of the approaches that came to light during the literature study in Section 10, did indeed rely on clustering instead of a supervised method. Unfortunately, clustering based approaches come with their very own set of demerits, as elaborated in Section 13.1.1. In summary, these approaches apply heuristics based on the idea that the software under analysis contains repetitive patterns, e.g., the application of sanitization functions between similar sources and sinks. Then, if such repetitive patterns can be observed multiple times, they are likely important and correct. Based on this, the rest of the code can then be checked for compliance with these observations. If, for instance, there is a dataflow between the same source and sink functions, where in most cases there would be a call to a sanitization function, it is likely that in this instance there should be one too. This approach may be better, in a sense that it does not rely on general vulnerability patterns that must hold true across all possible software projects, but instead learns from the project under analysis alone. At the same time, however, this comes with the drawback that the tool must be able to make a sufficient amount of observations to recognize a pattern as potentially important and correct. Therefore, more general patterns that the source code may not contain to the necessary degree remain unrecognized.

Finally, both approaches offer limited possibilities for users to make adaptations to the models, since retraining supervised models, such as deep neural networks, is difficult and requires a great amount of training samples. Unsupervised methods may be in a slightly better position, but still offer no comprehensive way of manipulating the model at hand.

The ability to customize a SAST tool to specific requirements is considered to be very important, which was also reflected during the expert interviews conducted in Section 5 [2, p. 40]. There, multiple interviewees

[18] https://git.qemu.org/?p=qemu.git;a=commit;h=4917cf44326a1bda2fd7f27303aff7a25ad86518

agreed that the possibility to define a custom rule set, is crucial. One of the experts stated in this context that SAST tools "must not be viewed as off-the-shelf products, but rather as frameworks, which need to be adjusted to personal requirements". Another interviewee also remarked that their team would select rules from a predefined set on a per project basis, again highlighting the importance of adjustable models. Another reason why adaptable models may be considered crucial in the development of ML-based SAST solutions, is the lack of suitable training datasets. This matter could be considered a chicken-or-egg type problem, since there cannot be models that generalize well without adequate datasets and vice versa, if there is no interest in ML-based SAST approaches, no effort will be spent on improving these datasets. Adaptable models may arguably mitigate this problem to some degree, since although the models may not be ideal during their rollout, they can be improved during their lifetime. This effect might be additionally reinforced, if feedback loops are employed that allow the users to retrain their model, based on previous and validated findings. Finally, another benefit that comes with the use of adaptable models, could be the usability of such approach. The example-driven nature may offer some benefits to the rule-based operation of other SAST solutions. One of the interviewees remarked on the subject that finding bugs using self-defined rules is difficult, which may be attributed to the high complexity of static analyses and the expert knowledge that is required for this reason. Using adaptable models within an ML-based SAST tool, does not require rules to be defined. Instead, it is sufficient for a user to provide a sample of unwanted behavior, or likewise, wanted behavior. Keeping these ideas in mind, for the development of the prototype the following considerations were made:

1. The tool should be able to reason on the basis of more general patterns that may be found across software projects. This ability is likely more prevalent in supervised models.
2. On the other hand, the tool should also be able to consider project-specific defects as well, a trait commonly found in unsupervised ML-SAST approaches, as mentioned in Section 13.1.1.
3. Linked to the latter attribute, is the ability to customize the tool to such needs, as reflected in literature and during the expert interviews.

Starting at the very basis, the prototype is designed in such a way that it is easily extensible. To this end, the prototype was largely implemented in the Python programming language, which according to the TIOBE[19] and the PYPL[20] index ranks top of the most popular programming languages in 2022, thus attaining a large user base. Furthermore, the Python programming language benefits from a large selection of machine learning libraries, such as the renowned SciKit-Learn[21] and the TensorFlow[22] libraries.

The extensibility and the framework character, mentioned during the interviews, was also considered in the prototype's architecture. The tool is organized in a clear and easily comprehensible pipeline structure that is composed of multiple phases. Each phase inherits from the same base class and must simply implement a single method that serves as the entry point. Phases may also be configured to require another phase to be run prior to its own executing. The prototype keeps track of the order of execution and ensures that all of these invariants are met. This allows users of the tool to write their own phases and experiment with different program representations, as well as machine learning algorithms.

The prototype is designed around the idea that a project is a simple directory, which serves as the main interface between the different phases, each of which may place files with the analysis results in the directory. There, consecutive phases can read the results and continue with their analysis. Initially, a project is defined by three items that the users must provide:

[19] https://www.tiobe.com/tiobe-index/
[20] https://pypl.github.io/PYPL.html
[21] https://scikit-learn.org/stable/
[22] https://www.tensorflow.org/

1. The source code that the user wishes to analyze, preferably within a direct subdirectory of the project directory.
2. A BASH script that handles all steps necessary, to compile the source code into a format that the first phase of the analysis requires. Along with the BASH script, a C/C++ entrypoint file needs to be provided which calls all APIs defined by the source code.
3. A configuration file in the YAML file format that specifies the analysis phases to be executed and their order, as well as the phase-specific parameters necessary.

With this defined, the prototype reads the project directory and wraps it into as a single class that stores the context, e.g., the analysis phases run or the phase-specific configuration. Afterwards, it executes the phases in order of their definition. Finally, when the last phase has finished its execution, the results may be presented to the user via an external front end. Since the prototype itself merely poses a framework, it does not dictate a specific output format, as the requirements to such are highly dependent on the analysis at hand. Figure 32 illustrates the basic working principle of the prototype. As can be seen, the software project to be analyzed is made up of the source code itself, the build script and a configuration file in the YAML-format. The prototype, controls the execution of the three phases called: *Graph Generation*, *Path Extraction* and the final *Analysis*. Finally, the result may be viewed and used to make adaptations to the models using a frontend, which is an external component, described in Section 13.2.6.

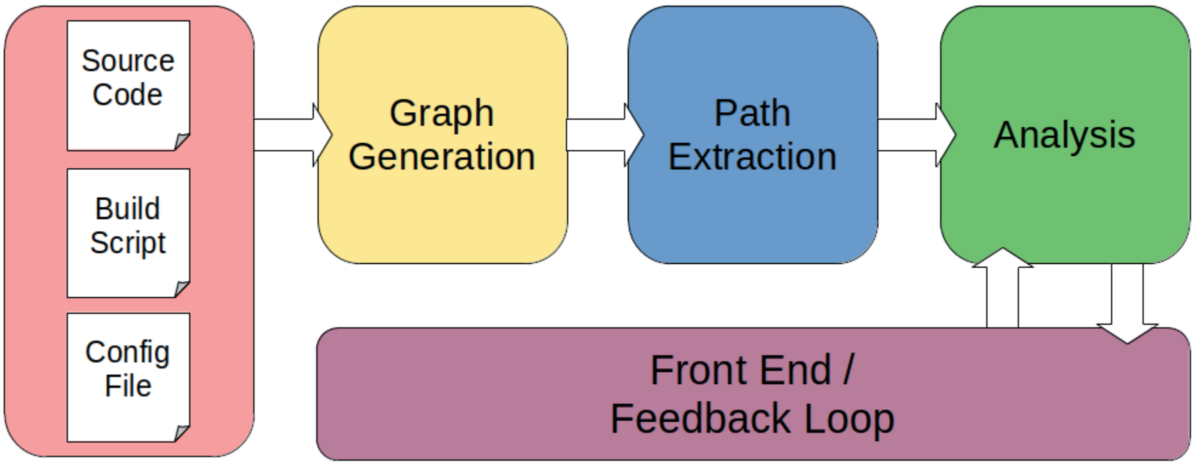


Figure 32: The basic working principle of the prototype.

13.2.2 Demonstration Pipeline

As elaborated before, the prototype is organized in individual phases. So as to demonstrate the capabilities of the prototype, a pipeline was devised that aims to detect control-flow related vulnerabilities. The choice to focus on the control flow was made, as it covers a lot of vulnerabilities, e.g., missing return value checks (CWE-252, CWE-690), NULL pointer dereferences (CWE-476) and if relying on the interprocedural control flow, also defects related to taint style problems that are reflected in the order of procedure calls. Furthermore, the control flow can be extracted from a program in a comparatively fast manner. Whereas building complex program representations, such as the aforementioned SDG or SVFG requires a considerable amount of computation time, extracting the control flow does so to a far lesser degree, being merely a perquisite for both representations. When comparing the different representations during some explorative tests that were conducted early into the implementation of the demonstration pipeline, it appeared that the interprocedural control flow seems to strike an excellent balance in terms of complexity and informative value. This balance may be responsible of allowing models to be efficiently learned on this type of program representation. Due to time constraints, it was not possible to evaluate these factors in a systematic fashion, leaving this task up for further research.

The interprocedural control flow graph (ICFG), on which the demonstration pipeline relies, comprises all control-flow related statements, including procedure calls and returns. In order to achieve precise results, the control flow cannot be extracted immediately from the program. Instead, it is necessary to conduct a PTA prior in order to resolve all possible function pointers. Function pointers are often found in C and C++ software, as they are extensively used in programming patterns that rely on call-back functions. As outlined in Section 13.1.3, finding suitable analysis tools that comprise a sound PTA are scarce. The SVF library was the only one that offered the necessary capabilities. In addition, SVF is able to generate multiple different kinds of program representations that could also be used if desired. These representations comprise the PTACG, VFG, SVFG and through an addition made while experimenting, a top-level dependency graph (TLDG) that is built from the LLVM-IR opcode. This graph links all statements in the code over their operands and return values.

13.2.3 Graph Generation Phase

The graph generation phase is the first in the pipeline depicted in Figure 32. It is a wrapper for the aforementioned SVF library, used to perform the PTA on the source code and to translate the program code into different graph representations. SVF works on the basis of LLVM bit code, an intermediate code representation that the LLVM compiler infrastructure uses to implement optimization passes that are run during compilation. This format is considered to be a partial form of static single assignment code, since it only covers top-level, but not address-taken variables [126]. The Clang compiler, which is built as a frontend upon the LLVM compiler infrastructure, emits this format and together with the tool “llvm-link”, a linker for LLVM bit code, forms a complete compiler-framework. To avoid that practitioners of the demonstration pipeline have to resort to manually translating and linking the source code, the WLLVM compiler wrapper[23] is used, which automates this process. The below Figure 33 shows this process in detail and corresponds to the equally named ”graph generation” element in the previous Figure 32.

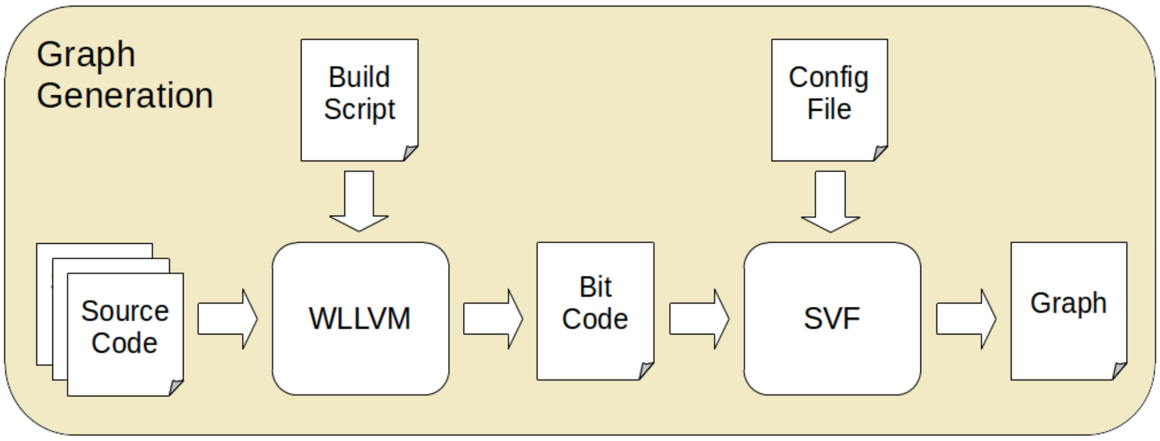


Figure 33: A schematic overview of the graph generation process.

While being a good choice from a technical perspective, SVF has the downside that the tool is very complex and the internal graph structures are only exposed through a C++ API for querying. Taking the statements from the interviews into account, where it was criticized that it was hard to detect bugs using self-defined rules, having to resort to a C++ API may be considered too complicated for most users. This is due to the fact that with C++ being a compiled language, single queries would have to be compiled before being executed. Debugging queries would likely be also rather demanding and the library alone would offer no way of persisting graphs for later use as they reside in main memory. To alleviate this, it was decided to make use of the Neo4J graph database in form of an intermediate phase. The Neo4J graph database allows the stored data to be queried as a graph traversals and is therefore optimal for this use-case scenario. The

[23] https://github.com/travitch/whole-program-llvm

Neo4J database in particular uses the Cypher query language for property graphs, which is comprehensible and does not require to be compiled first [127]. Furthermore, the Neo4J database itself is very extensible and allows for the development of user defined plugins in the Java programming language. Besides this, extensive libraries are readily available, which add useful features to the database, making the extraction of subgraphs for the later analysis very comfortable. Lastly, in contrast to the pure C++ API, the Neo4J database allows the graphs to be viewed through a client. This allows for queries to be easily developed and debugged before deployment and has the additional bonus of helping users visualize and understand the program they are analyzing. In order to be able to load the different graph representations generated by SVF into the Neo4J graph database, it is necessary to first export these into a compatible format. Conveniently, Neo4J allows for bulk imports to be made from simple CSV formatted files. Thus, using the SVF library, a simple wrapper tool was devised that allows the user to generate the aforementioned program representation graphs, supported by SVF. The graphs are then written to their respective CSV files, depicted as the *Graph* element in the above Figure 33.

Finally, the entire compilation process using the WLLVM tool, the graph generation using SVF and the export into CSV-formatted files that are readable by the Neo4J graph database are wrapped as a single Docker container. Docker is the name of a virtualization software that runs on the operating system level, meaning that while sharing the same kernel, all applications running in different containers are otherwise isolated from each other [128]. Not only does this help bundle the required software in a portable way, but it also means that every software project is being built and analyzed in a fresh and controlled environment. Since every software project has specific requirements as to the steps necessary for building the software or relies on external libraries that must be installed prior to its compilation, users are obliged to provide a shell script that takes care of these steps. To this end, the users have access to the container by means of a single BASH script, which can be freely customized. This script is called every time the container is executed by the prototype. Further, SVF-specific settings are exposed through the aforementioned YAML-file and are passed to SVF as command line arguments.

13.2.4 Path Extraction Phase

As explained in Section 13.2.3, the C++ API made available by the SVF tool alone was insufficient, which is why the program representation graphs generated prior are loaded into a Neo4J graph database. Similar to the step before, this graph database makes it easy to dissect the graph into smaller components that can then be processed by ensuing ML-based analysis phases. Similar to the graph generation phase that was encapsulated into as single docker container, this phase is also executed within such container. The corresponding docker image is made readily available by the Neo4j project and takes care of the otherwise complicated configuration of the database service, hence this section focuses on the additional plugins used and the extracted data structures during this phase instead. The below Figure 34 depicts this phase visually and corresponds to the equally named *Path Extraction* element in the previous Figure 32. As can be seen, the phase begins with the output from the last phase, by importing the generated graphs into the database.

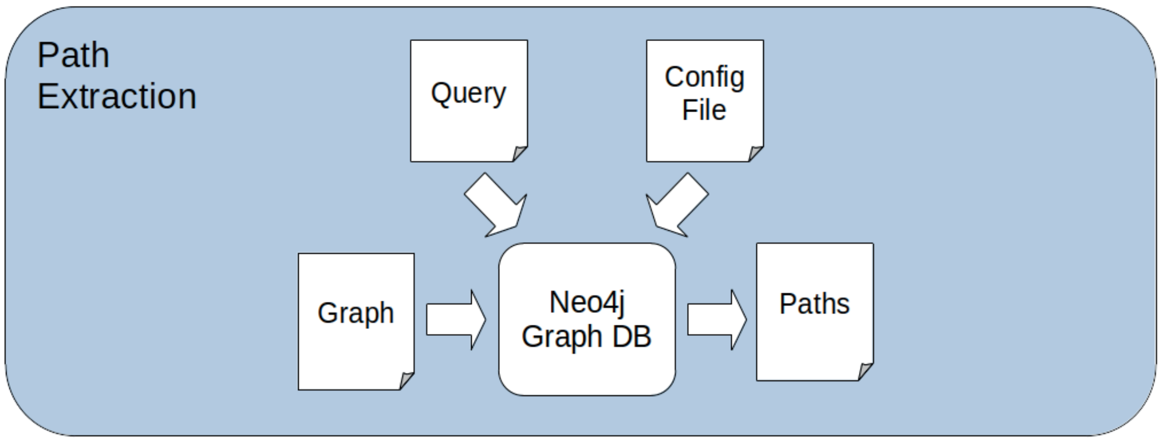


Figure 34: A schematic overview of the path extraction phase of the prototype.

Since the program graphs have special traits, it was decided that it would be best to collect common utility functions and queries into a single plugin for the Neo4j database. Neo4j offers a comprehensible API that allows for the development of such plugins in the Java programming language. The functions that are defined there are then callable from within normal Cypher queries [129]. Aside from some convenience functions, e.g., one that returns a list of all procedures within the graphs, this opportunity was mostly used to make further changes to the graphs after the generation phase. Most importantly, a function was created that adds additional edges between all corresponding nodes between the different graphs. This means that one can easily switch between graphs within a single query and is not confined to using a single graph only when extracting subgraphs from the database.

As explained in Section 13.1.2, it is extremely difficult to embed complex graph structures into a vector space and the available methods to this end are scarce. The only applicable method found, during first attempts, was the Graph2Vec method, which in this particular case, performed underwhelming. One could argue that GNNs do not suffer from this limitation, as they are learned directly on the graphs. For the demonstration pipeline discussed here, it was decided against the use of this type of neural network, due to the distinct lack of explanation methods available today, see 11.3. If one were to move away from the idea of embedding complex subgraphs and instead rely on simple paths, this problem can be circumvented entirely. Although the embedding of complex subgraphs still remains an important topic, finding an appropriate solution was simply not within reach in the time frame of this project. It generally appears that a lot of research is necessary in this regard, if a satisfactory solution can be devised at all. As for the demonstration pipeline, it was thus decided to discard the idea of embedding complex subgraphs entirely and instead embed single paths from the ICFG instead. To do so, all procedures that are contained within the ICFG are found. From there on, all unique paths reaching from the entry to the exit node of the procedure, are explored and the nodes visited along the way are extracted in the order of traversal. Figure 35 lists the Cypher query that is executed to this end, corresponding to the element called “Query” in the above Figure 34.

```
CALL mlsast.util.listProcedures()
    YIELD str AS f

WITH f
MATCH (n:FunEntryICFGNode {func_name: f})

WITH f, n
MATCH (m:FunExitICFGNode {func_name: f})

WITH f, n, collect(m) AS exits
CALL apoc.path.expandConfig(n, {
    relationshipFilter: "icfg>",
    terminatorNodes: exits,
    uniqueness: "NODE_LEVEL",
    maxLevel: 75
})

YIELD path
RETURN f, path;
```

Figure 35: The Cypher query that is used to extract all paths within the ICFG for each procedure.

As can be seen, using the aforementioned MLSAST-plugin for the Neo4j database, a list of all procedures within the database is acquired. Then for each procedure, the entry and exit nodes are found and the “expandConfig”-function is called to extract all unique paths between these nodes. This function is part of the APOC-library, which the Neo4j provides. It extends the database with useful functions, such as this one. The “expandConfig”-function allows for the expansion of paths and offers fine-grained configuration options. In this instance, the path expansion is configured to follow ICFG-type edges only in the forward direction, where for each path yielded, the nodes are guaranteed to be unique on the same level, meaning that loops are effectively prevented from being extracted. The maximum depth of these traversals has been limited to 75 nodes, as anything larger would be likely too intense computationally wise. The path extraction method described here, is also shown in Figure 36. The graph denoted with the number **I** depicts the full ICFG of a procedure. Following this approach, there are three unique ways by which the graph can be traversed from the entry to the exit node, the first being demonstrated in graph **II**.

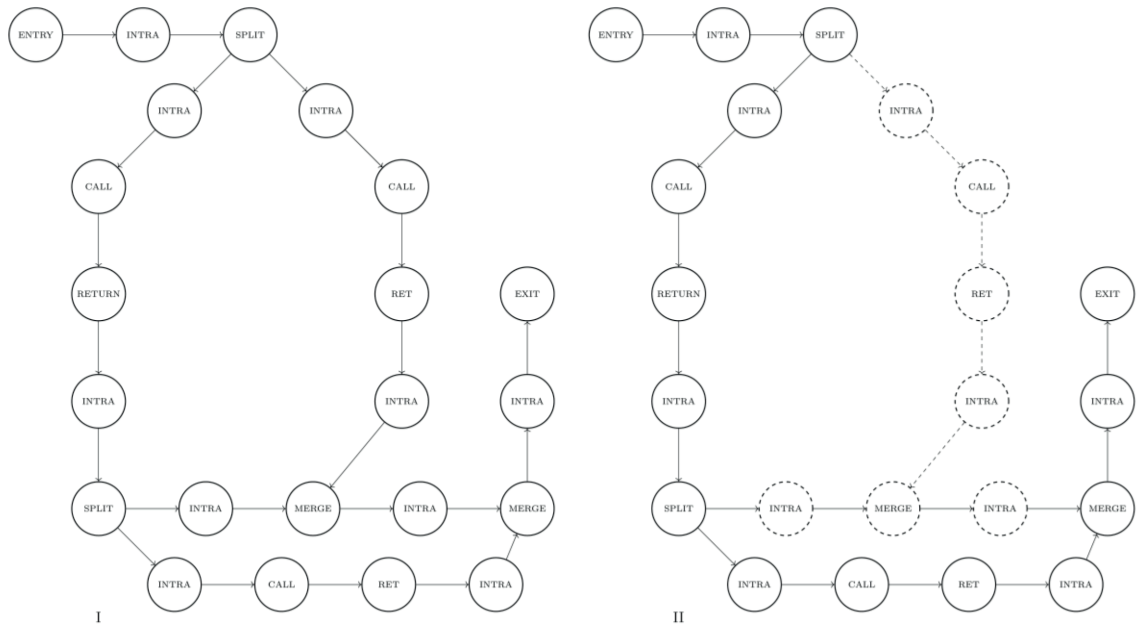


Figure 36: A simplified example of the path extraction method. The ICFG denoted with the number I is the full graph, II shows one out of three possible traversals from the entry to the exit node.

The nodes traversed on this path are drawn using solid lines, whereas the nodes that do not lie on this path are drawn using dashed lines. Moving on, there are two other ways, by which the entry node is connected to the exit node. Figure 37 depicts these paths in graphs **III** and **IV**. It must also be noted that the ICFG is a directed graph, hence in this example there are only three ways to traverse the graph from the entry node to the exit node. Therefore, loops are not possible in this simplified example. In general, however, loops are possible also in directed graphs. As for the prototype, the *expandConfig*-function was parametrized to not unroll loops, but instead to only traverse them once. This was done to keep the extracted paths as short as possible.

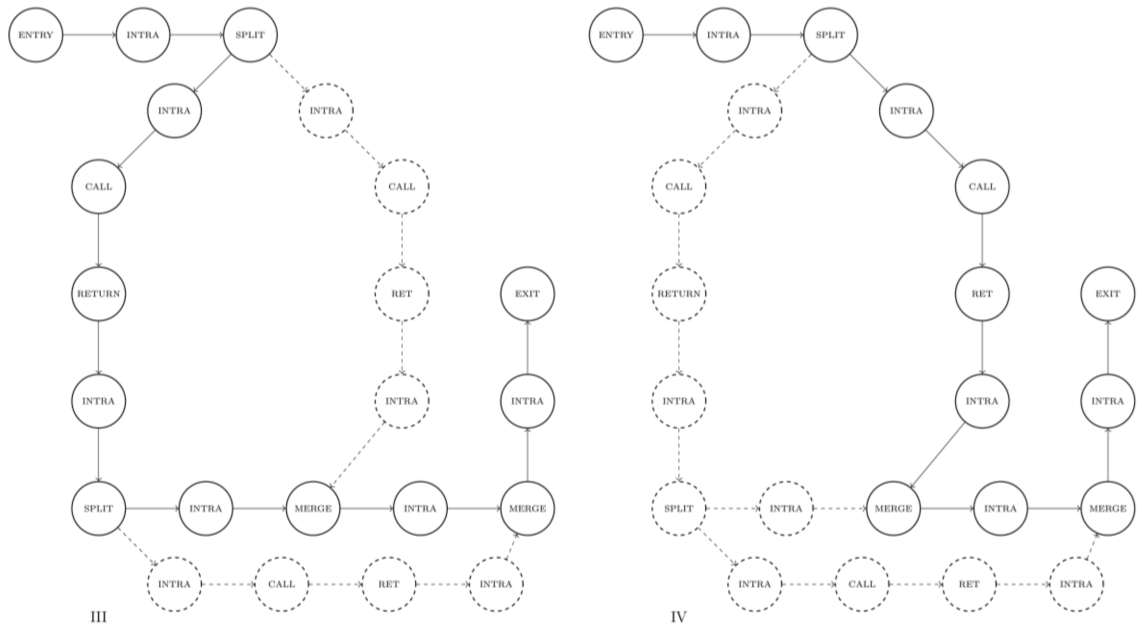


Figure 37: The other possible paths, by which the graph depicted in Figure 36 may be traversed.

Since the demonstration pipeline is solely based on the ICFG, the database queries devised to dissect the graph into smaller components are specific to this graph and the problems that are detectable using this type of program representation. Not only can the SVF-library generate the ICFG of a program, it is also capable of generating the SVFG, which captures the dataflow of a program and is thus ideal for the detection of taint-style vulnerabilities. Here too, paths could be extracted, however, this time between calls to a source and a sink function. It would also be conceivable to use the PTACG that SVF can generate in order to find other types of bugs that manifest themselves in the order of procedure calls. This is just to offer a glimpse into the types of analyses that the prototype could likely be extended with under manageable effort. This is up to further research, as due to time constraints, all energy was focused on this one demonstration pipeline only.

13.2.5 Detection Phase

Coming back to the initial premise that a SAST tool must be extensible and configurable, the detection phase does not pose an exception. In the field of machine learning this is difficult, as models that have been trained once are fixed. While adaptations are possible, they require a large number of additional training samples and retraining them is not easily accessible to end users. To allow for this to be possible and in order to strike a sensible compromise between the merits and the demerits of supervised and unsupervised approaches, the solution described in Sections 13.2.5.1 and 13.2.5.2 was devised. As before this section picks up where the previous phase from Section Section 13.2.4 has left off. It uses the paths that were previously extracted from the ICFG and compares them against known defective paths from the Juliet test suite. Figure provides an overview over this process.

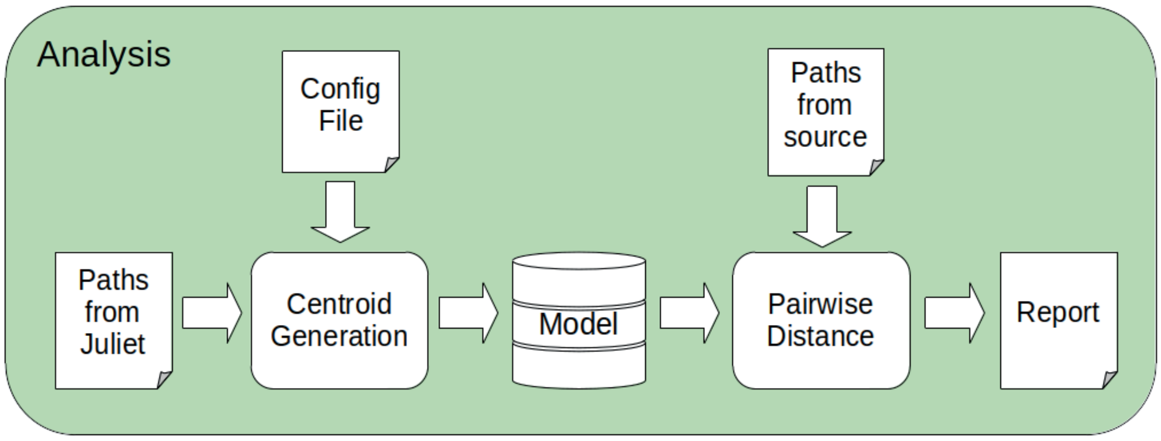


Figure 38: General overview of the Analysis phase.

13.2.5.1 Basic Working Principles

First of all, all supervised approaches, known from the literature study, were deemed far too limited in terms of adaptability, see also Section 13.2.1. Although unsupervised approaches appeared to be better in terms of adaptability, they were far from ideal either. This kind of approach also suffers from the drawback that it only considers project-intrinsic bugs. Both approaches may be combined, however, if instead using a randomly initialized clustering algorithm, the algorithm would be initialized from a known ground truth. This idea is based on the observations that the ICFG paths extracted in the prior analysis phase, see Section 13.2.4, demonstrate a crucial property: When comparing paths extracted from a vulnerable implementation against those from the benign implementation of functionally similar code, they are often discernible. This may be reflected in the number of nodes of the affected path, the type or opcode of these nodes and finally the order of nodes (by type or opcode). For example, in case of some missing NULL-pointer check w.r.t. the return values of a procedure, the resulting path would be shorter and lack important instructions. Figure 39 shows two implementations that highlight this behavior. Both pieces of code make a call to the library function “malloc()” in order to allocate heap memory for a single character value. As per manual, if the allocation fails, the “malloc()” function returns a NULL-pointer. Performing a NULL-pointer check after each call is thus paramount, since the omission could potentially lead to memory being read or written at the invalid address of 0, causing a segmentation fault to be triggered. The implementation to the left, does perform a NULL-pointer check, hence showcasing the correct practice. The one to the right does not and may thus cause the program to crash, if “malloc()” returns with a NULL-pointer.

```c
// Benign implementation
int main(void)
{
    char* buf =
        malloc(sizeof(char));
    if (buf != NULL) {
        *buf = 'a';
        return 0;
    }

    return 1;
}
```

```c
// Defective Implementation
int main(void)
{
    char* buf =
        malloc(sizeof(char));
    *buf = 'a';
    return 0;
}
```

Figure 39: Two implementations of semantically similar behavior. The code on the left is the benign implementation as it performs a NULL-pointer check after calling malloc(). The implementation of the right does not and thus might cause a segmentation fault.

Using SVF and extracting the individual paths for each function, as described in Sections 13.2.3 and 13.2.4, the following ICFG paths may be obtained for both implementations. Figure 40 shows these paths, where path I is the correct implementation and path II is the defective implementation from the examples in Figure 39. It must be noted that these ICFG paths are somewhat simplified, as simple instruction nodes have been omitted for the sake of brevity. Also notice that the additional edge in path I, directly connecting nodes 3 and n-1, would not be present in actuality, since all nodes on a path have an ingoing and outgoing degree of 1. It has merely been added to this example to highlight the fact that for the correct implementation multiple paths in the control flow exist.

Path **I** starts with the ENTRY node to the function and then immediately performs the call to malloc, depicted by nodes 1 and 2. Then the control flow diverges in node 3, in accordance to line 6 in Figure 39, depending on whether the pointer variable “buf” points to NULL or not. This is represented by the dashed place holder node, which comprises several simple instruction nodes that do not excerpt any direct influence on the control flow. Then, in node n-1, the paths merge again and the function exits in node n.

Path **II** follows the same behavior up until node 3. Since this is the only path that connects the entry and the exit node within this function, there are neither SPLIT nor MERGE nodes present. Instead, other simple instructions follow, illustrated again by a dashed place holder node, marked [3 … n-2]. This circumstance renders these paths highly discernible, by length and by type. Section 13.1.2 and 13.2.3 already elaborated on the difficulties of embedding entire subgraphs, which is why the extraction and analysis of paths was chosen instead for this demonstration pipeline. Still, so as to allow for machine learning techniques to be applicable, it is necessary to embed these paths into a metric vector space. Figure 41 exemplifies the method employed to this end.

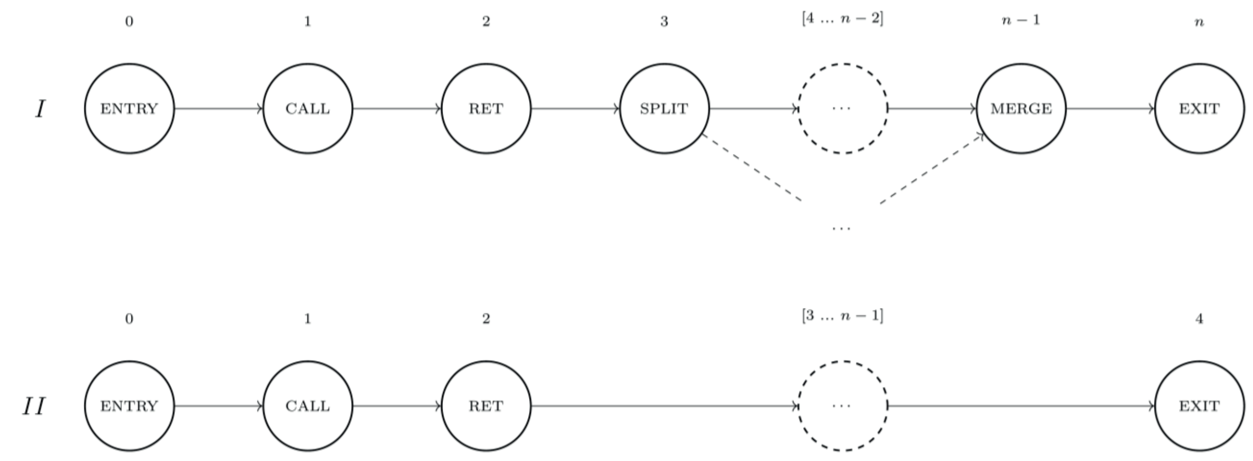


Figure 40: Two simplified ICFG paths that were extracted from semantically similar procedures. Path I shows the benign implementation, path II the defective one.

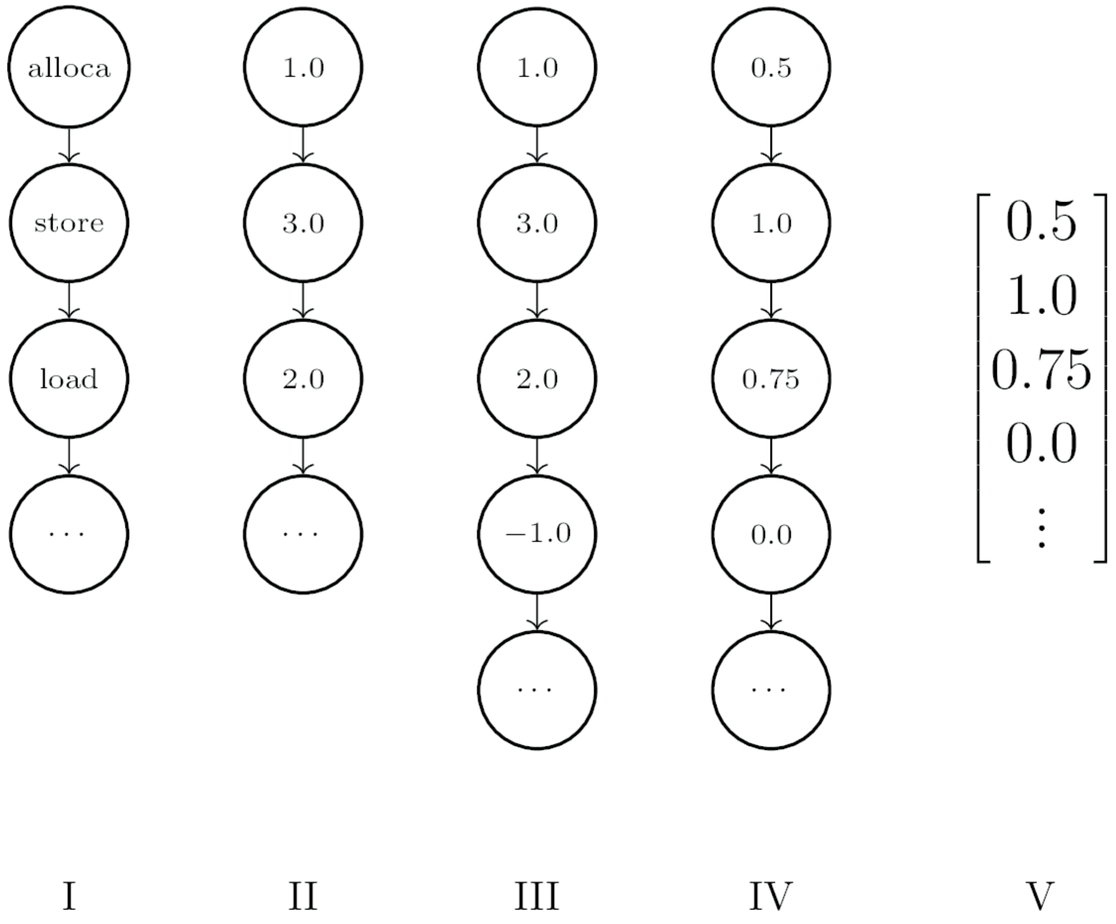


Figure 41: The steps employed in order to embed the individual ICFG paths into a latent vector space.

Starting with an ICFG path, marked as **I** in the illustration, the first step employed is the mapping of the nodes onto a set of real-valued numbers with respect to their opcode in step **II**. Then, since paths are likely to excerpt different lengths over all paths extracted from the program, they must be padded to fit the length of the longest path, see step **III**. Finally, the last step comprises the normalization of the paths onto values between 0.0 and 1.0, visible in step **IV**. This was shown to be beneficial in terms of discernibility. It is then possible to understand these paths as real-valued vectors if each index within these paths is assigned a fixed dimension in a vector with the dimensionality of the longest path, marked as **V**.

13.2.5.2 Model Generation

Using the aforementioned methods to process a number of benign and vulnerable program paths, a vector space is spanned, where each path is a point in that space, giving rise to the possibility of applying machine learning techniques. To this end, a method was devised that allows for quick and explicit adaptations to be made to the models. This method will be will be elaborated in this section. Figure 42 provides an overview of the entire process, for reference purposes.

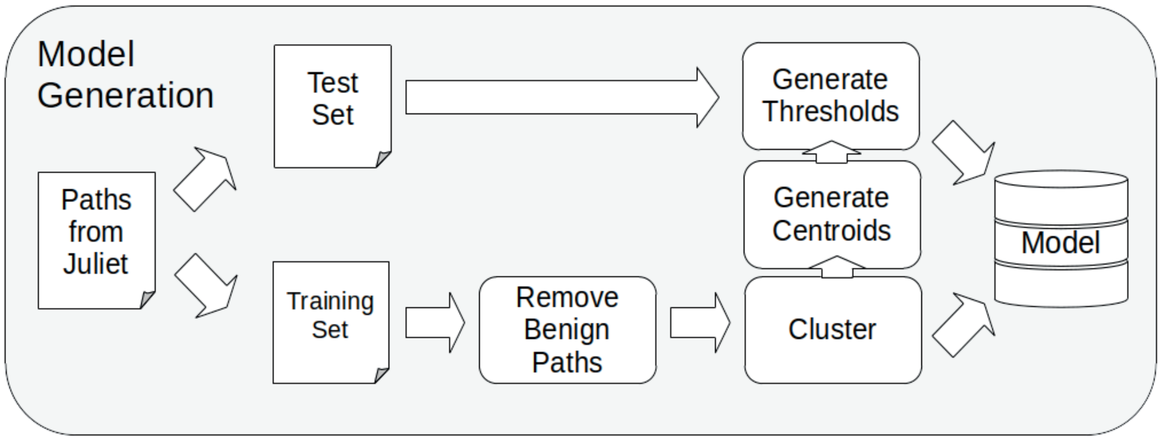


Figure 42: Schematic overview of the model generation process, which is part of the detection phase, depicted in the previous Figure 38.

Firstly, in order to generate models, it is necessary to have a ground truth, where it is known beforehand, which parts of the code are defective and which are benign. Since datasets based on real software projects may suffer from label noise issues, it was decided to use the Juliet test suite. This dataset has the drawback of being synthetic, thus not being entirely representative of real vulnerabilities found in actual code. Nevertheless, it may be argued that due to the high adaptability of the approach implemented in this prototype, this circumstance poses less of a negative impact than a skewed ground truth. More realistic datasets, generated from real software projects, may suffer from this detriment, due to the fact that the class of benign samples is usually taken from real software projects as well. This is done under the assumption that if no vulnerability is known for a given code sample, then it must be safe. In reality, however, it is likely that the samples yielded this way, possibly contain a number of unknown vulnerabilities, yet to be discovered. Hence, label noise is introduced, which may pose a negative impact on the model's predictive performance. As for the prototype, the models can be adapted quickly, so that it is possible to use prior findings of the prototype to refine the models to better fit real-world vulnerabilities. Another benefit of the Juliet test suite lies in the fact that it can be compiled with standard compilers, such as the Clang compiler, as each individual testcase forms a self-contained program. This is a key requirement, in order to be able to make use of the SVF-tool for the generation of the ICFG, as elaborated in Section 13.2.3. Otherwise, it would have been necessary to resort to the Joern tool, which is able to process incomplete code, thanks to its fuzzy parsing capabilities. Its downside, however, is the lack of any option to perform a points-to analysis, as stated in Section 13.1.3. Finally, the modularity of the Juliet test suite has one last merit: As it is neatly divided into individual CWE classes, i.e., types of vulnerabilities, the Juliet test suite allows for the generation of not one, but several models that represent each of the CWE classes. Thus, the code can be tested against each model, or therefore CWE-class, in a separate fashion. The models that achieved the highest score could then be used to determine the most likely type of defect present in the code, providing the user with some more refined information. In the following, the basic principles by which the models are generated and used to detect defective code, are explained in detail. Furthermore, the observations that influenced the design of the prototype, will be elaborated.

In order to generate the models from the Juliet test suite, every CWE-class is processed individually. Using the test cases from the CWE-690 class as an example, firstly, the aforementioned extraction and embedding technique are applied. This is depicted as the *Paths from Juliet* element in the above Figure 42. Then the resulting datapoints are divided into train and test splits of equal sizes. Continuing with the training set for the time being, it can be observed that the paths form some loose clusters, as is visible in Figure 43. Here each datapoint represents a single path in the vector space, where green paths are benign and red paths are defective.

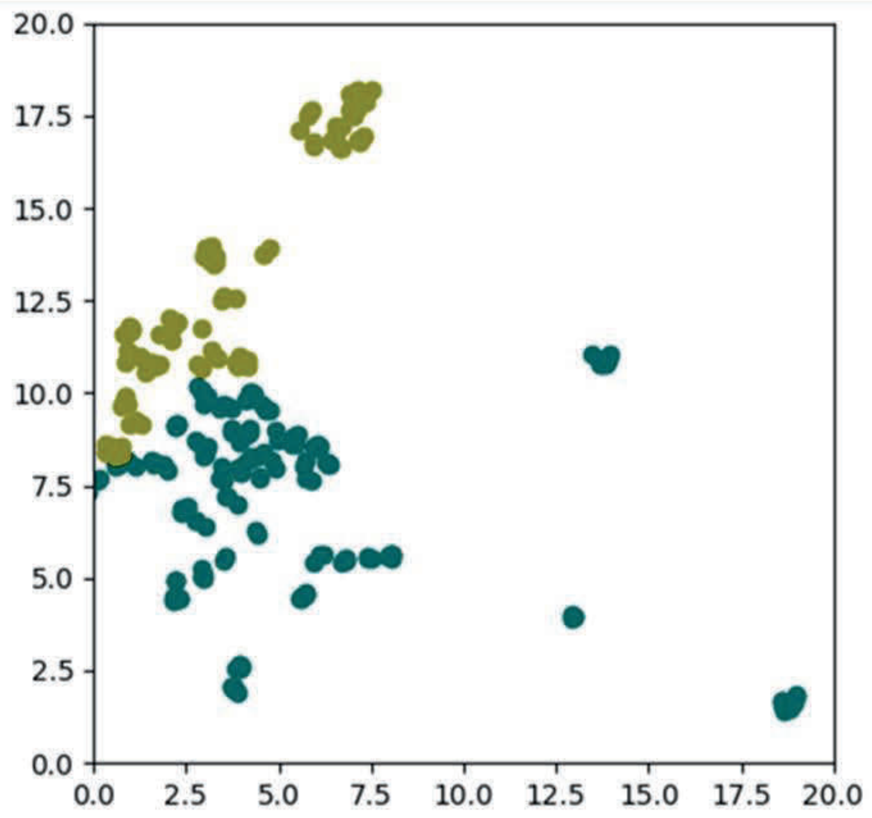


Figure 43: Benign (blue) and defective (yellow) paths of the CWE-690 type samples from the Juliet test suite. Embedded using the explained methods and reduced in dimensionality using the UMAP-method.

It must also be noted that the actual dimensionality of these paths is naturally much higher. So as to make the basic working principles observable, the Uniform Manifold Approximation and Projection for Dimension Reduction (UMAP) method has been applied to the paths [130]. All subsequent operations shown, were applied on top of the reduced paths, hence the example is not entirely representative of the actual affairs in terms of scale or clusters, but it serves well for elucidation purposes. Although Figure 43 depicts both, benign and defective paths within the same vector space, the prototype only ever uses one out of both within a single model. This is because it could be observed that mixing benign and defective paths, or therefore even paths from different CWE-classes, would only lead to the models being confused. Thus, the following examples only focus on the defective paths of the Juliet test suite's CWE-690 type samples, visible in Figure 44.

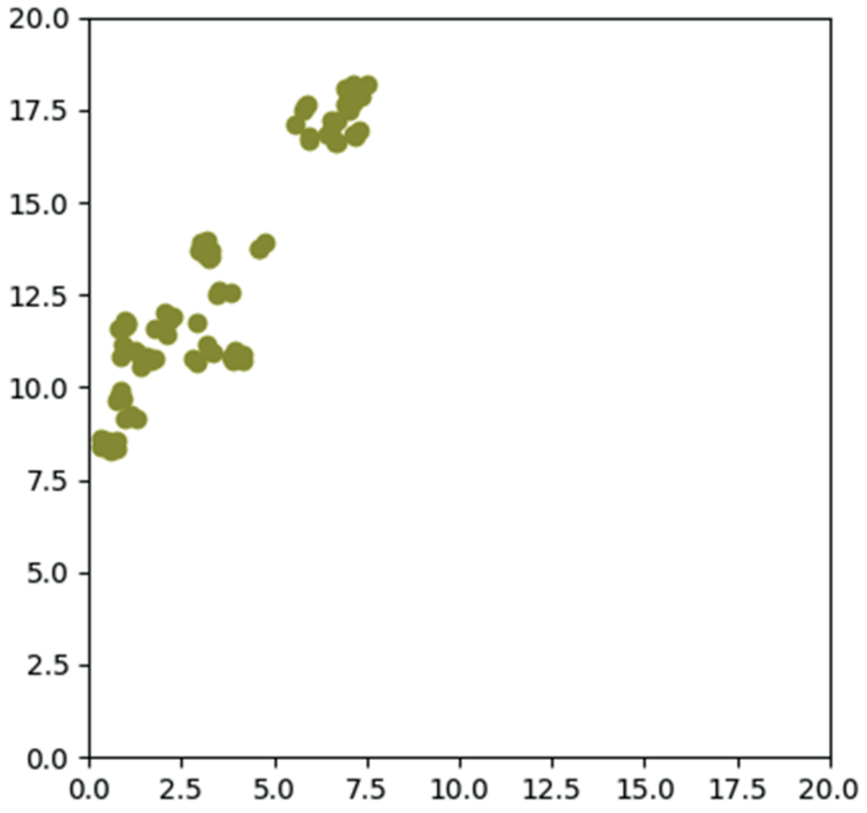


Figure 44: The same paths from before, but the benign ones have been removed as the prototype only considers one type of behavior at once.

As mentioned earlier, it can be observed that the paths form some clusters within the vector space, despite being from the same CWE-class and only focusing on the defective paths. This is because the test cases are not necessarily entirely homogeneous within themselves. So as to further isolate specific behaviors, it was shown to be highly beneficial to first use a clustering algorithm, to further disintegrate the paths into individual behaviors. To this end, the K-Means algorithm was used and the area under the curve (AUC) was measured under a varying parametrization for the algorithm across all cases. This way, using a value k=8 was determined to be the best parametrization in this example, leading to good results in terms of the average AUC across all cases. Values higher than 13 only showed diminishing results and in few cases even worse results than for a lower K-value. Furthermore, it was feared that while showing improvements when tested against the Juliet test suite, increasing the number of clusters too much, would come at a loss of generalizability, i.e., lead to an overfitting effect. Alternatively, an agglomerative clustering algorithm might be used as well. During experimentation, there were no differences noticeable between both algorithms. Figure 45 depicts the clustered paths from before, where each cluster was highlighted with a different color.

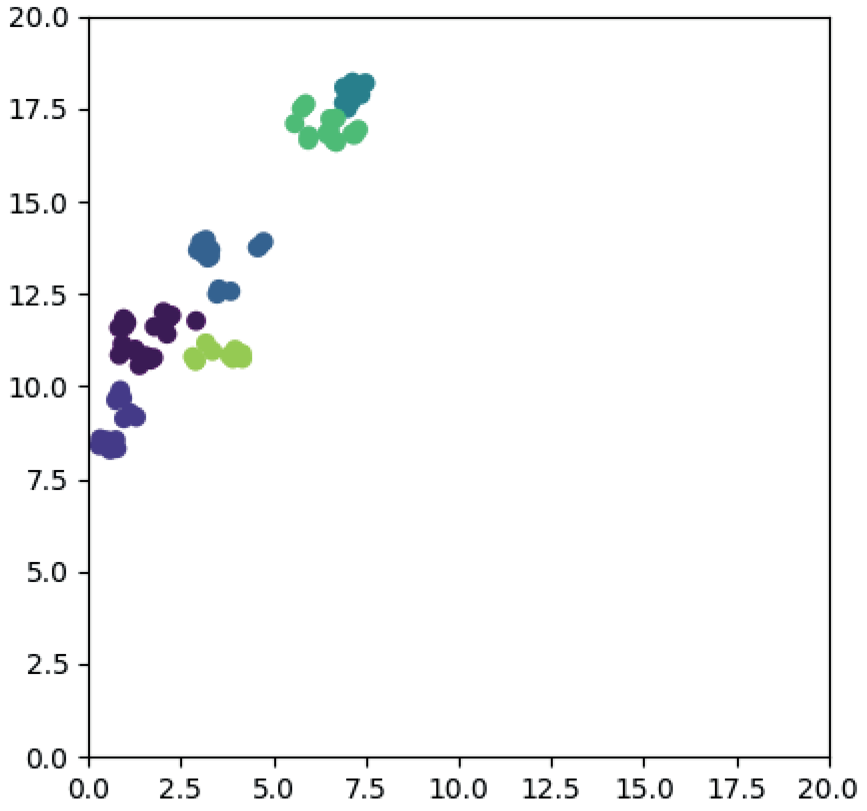


Figure 45: The defective paths of the CWE-690 class of the Juliet test suite, clustered using the K-Means algorithm parameterized with a K of 8.

Each of the clusters is now indicative of some specific variation of the defective behavior, defined by the paths from the Juliet test suites samples of the CWE-class 690. This trait can be used in the ensuing steps, if the geometric centers are formed for each cluster. They can be comprehended as the essence of some specific behavior. Thus, these geometric centers, also called centroids, are the first of two parts that the models are comprised of. The idea is that new and unknown paths - from the software that is being analyzed - may be compared against the clusters to determine if they are defective. Paths that are closer in the vector space to the defective centroids are more likely to be defective than those further away. Figure 46 shows the geometric centers of the defective paths, found in the Juliet test suite's CWE-690 class test cases. This step is depicted as the *Generate Centroids* element in the previous Figure 42.

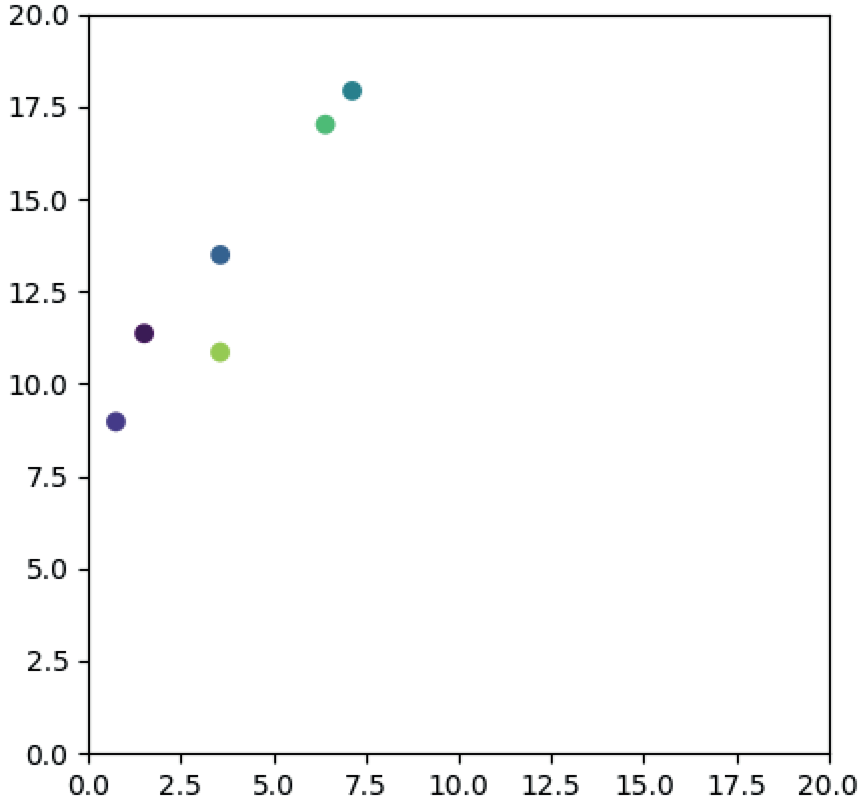


Figure 46: The geometric centers of the clusters depicted in Figure 45.

So far there are only centroids that are representative of some behavior. In order to come to a decision for each path analyzed, whether it is likely to be defective or not, there needs to be some threshold, beyond which a path would be considered benign, i.e., a safe distance. Of course, it would be a possibility to choose this value arbitrarily, but this is difficult to get right. Therefore, a technique was devised that adjusts this threshold parameter for each model automatically. However, since some practitioners may wish to make manual adjustments, the prototype allows for the threshold to be moved by a user defined offset. The problem of finding an ideal threshold was modeled as an optimization problem where the test split from earlier is employed and the distance between every path from the set to each of the centroids is calculated. It can be observed that benign paths on average showcase a higher distance to a defective centroid than a defective one to a defective centroid. Vice versa, a defective path, on average, has a higher distance a benign centroid than a benign path. In most cases, there is some overlap, unfortunately, meaning that benign and defective paths are equally far away from the centroids in some instances. In the case of the defective centroids, choosing a small threshold would then result in a decreased number of false positives, but an increased number of false negatives as well. A large threshold, on the other hand, would lead to the opposite and result in a high number of false positives, but therefore a decreased number of false negatives. The question now is: At what point is the model's predictive performance the highest? This point can be approximated by the computation of the Receiver Operating Characteristic (ROC). The ROC plots a classifier's true positive rate (TPR) against its false positive rate (FPR) while increasing the threshold, in this case the radius around the centroids. At each increase, the TPR and FPR are taken and its geometric mean is calculated, given by $\sqrt{(\mathrm{TPR} \cdot (1 - \mathrm{FPR}))}$. The optimal threshold is reached, when the geometric mean reaches its maximum. Figure 47 visualizes these circumstances. Depicted left is the histogram that plots the distances of all benign and defective paths from the test split against the defective centroids generated from the training split. The x-axis spans the minimum and maximum distance measured over all paths, whereas the y-axis depicts the number of observed paths at these distances. The blue paths are the defective paths, closer to the centroids, the orange ones are the benign paths that are further away. The grey area marks the overlap discussed before. This example exhibits a mediocre discernibility. To the right of the histogram, is the corresponding ROC curve that is used to determine the best threshold. In this example, a threshold of 1.9 would be considered ideal.

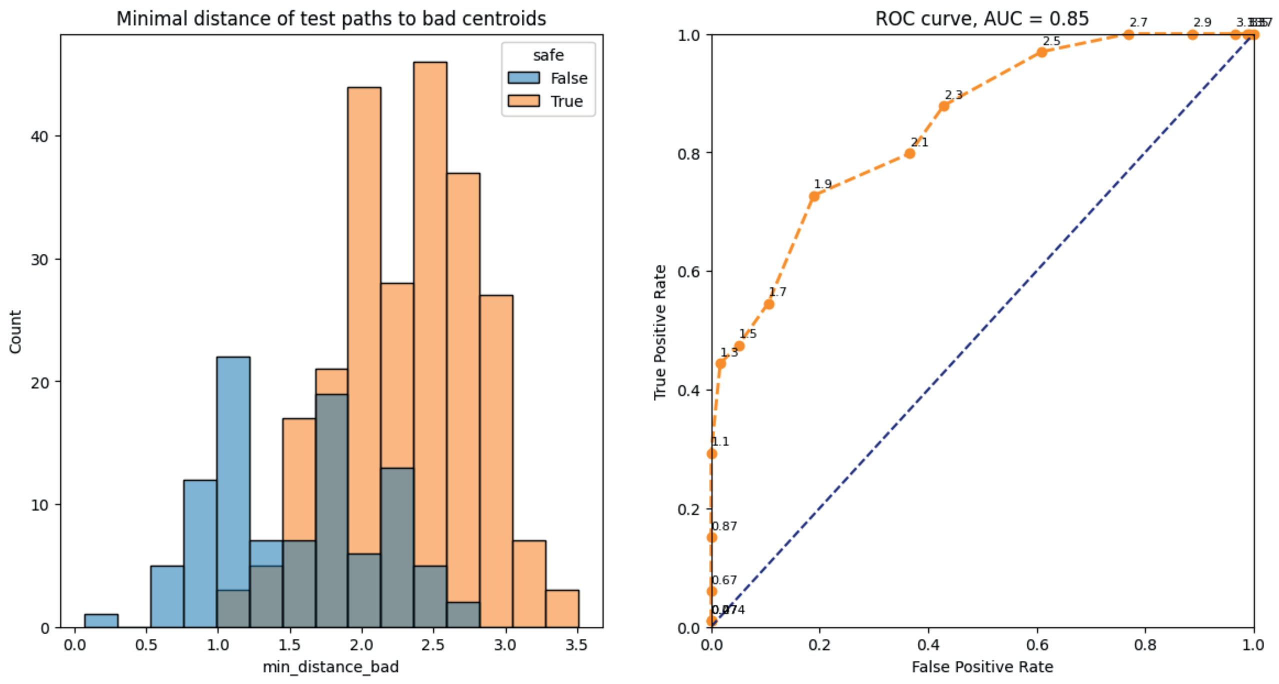


Figure 47: The histogram on the left depicts the distances measured between the paths in the test split and the defective centroids for the CWE-class 690. Seen on the right, is the associated ROC curve, where the labeled datapoints correspond to the threshold for this ROC.

Finally, with these thresholds at hand, the model is complete and can be used to predict the probability of new and unknown paths from the software under analysis, as shown in Figure 48. Here the same, defective centroids of the CWE-class 690 are depicted as before, together with their optimal thresholds, marked by the red circles around them. For each new path, the distance is now measured and it is determined if they lie within the threshold of the centroids. Paths that do are marked as red diagonal crosses and are more likely to show some kind of defect and are reported to the user. Those that lie outside the thresholds are less likely to be defective and are represented by the green crosses in the figure. The entire procedure described is repeated for every CWE-class of the Juliet test suite, yielding an individual model in each case.

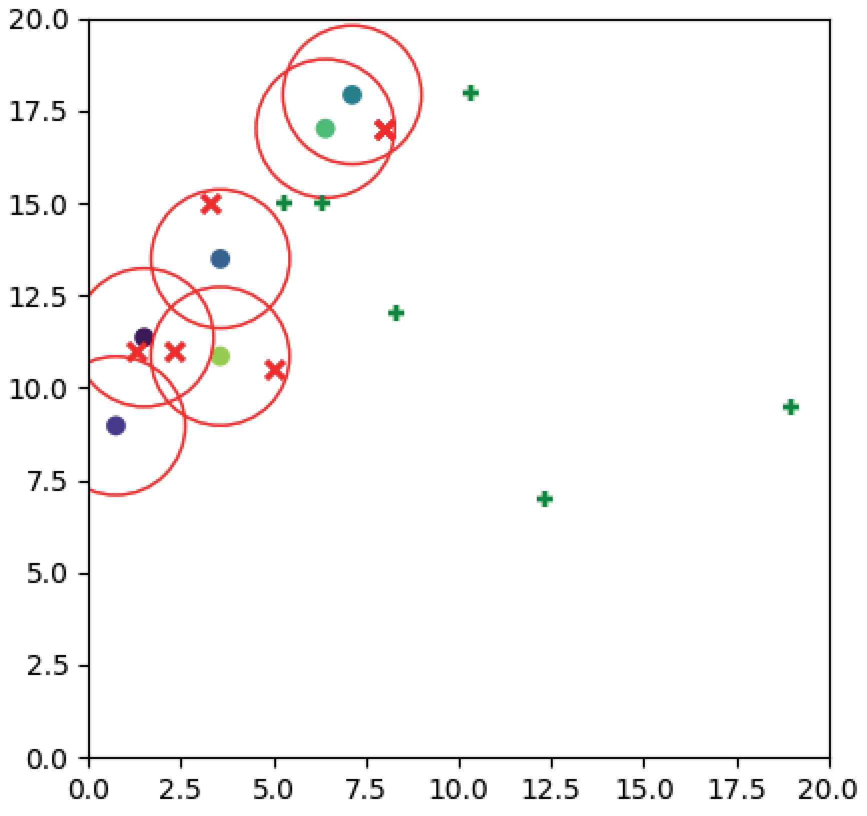


Figure 48: The position of unknown paths is observed in the vector space alongside the centroids and their thresholds. The red circles mark the aforementioned thresholds. Whereas saltires mark the potentially defective paths that fall within the thresholds, Greek crosses mark those that lie outside and are considered safe.

As mentioned, using separate models for each individual case, i.e., single CWE classes, appears to offer the additional advantage of providing more detailed information about the type of defect present. Whereas a binary classification model that was trained on the entirety of all possible defects is only capable of classifying code as either benign or vulnerable, testing the same code against a number of fine-grained models offers a much more refined result. Although being common practice with conventional SAST tools, most of the machine learning-based approaches that were brought to the fore during the literature study in Section 10, did not have the capability to discern between multiple types of vulnerabilities. It could be systematically determined that the different models that were created for each specific case generally showcase lower internal than external distances, thus validating the approach. To this end, the ICFGs were generated for every single test case within the Juliet test suite and the paths extracted as usual. The resulting paths were first partitioned by their CWE class and then partitioned for a second time, into benign and defective paths. Following this, for each CWE, paths were randomly sampled and assigned to equally sized sets of benign and defective test and training data sets, i.e., a total of four sets. Then, for both, the benign and defective training data sets, centroids were generated and both test datasets were reserved for later use.

Next, a baseline was established by measuring the average pairwise Euclidean distance, between every benign centroid and every benign path of the same CWE and every defective centroid and defective path of the same CWE. This way, for each CWE an average internal distance was computed for the benign and the defective paths. The assumption was now that if the same experiment was repeated, but this time comparing the centroids with paths from other CWE classes, the average distance would be higher relative to the baseline. Therefore, again for each CWE the centroids were taken, but instead of using the paths from the test data set of the same CWE, paths from all other CWE classes were randomly sampled. The total number of paths that were sampled, matched the number of paths of the original (internal) test dataset. Again, this was done for the benign and the defective paths separately. Figure 49 depicts the results of these experiments. As can be seen, there is a noticeable difference in terms of the average distance. To exclude the possibility that these results were produced by pure chance, the experiments were repeated a total of eight times.

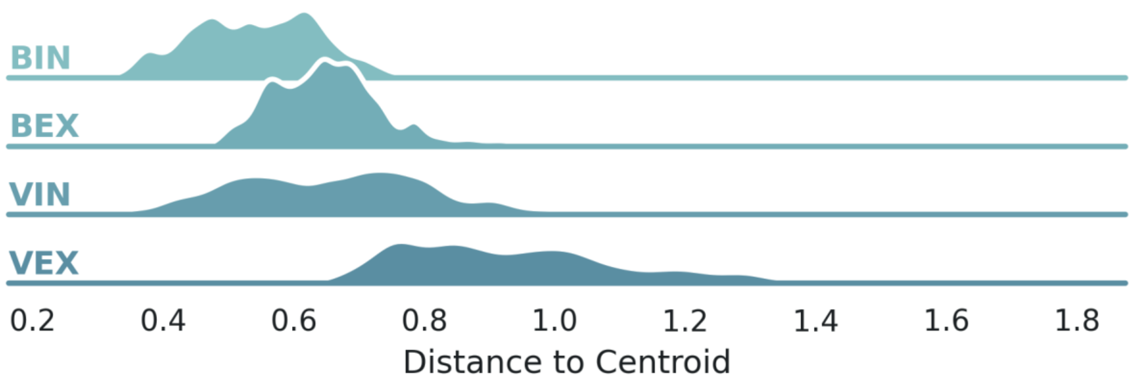


Figure 49: Average Distances between centroids and paths for each CWE class in the Juliet test suite.

Compared were benign centroids against benign paths from the same class (BIN), benign centroids against benign paths from other classes (BEX), vulnerable centroids against vulnerable paths from the same class (VIN) and finally, vulnerable centroids against vulnerable paths from other classes (VEX). As can be clearly seen, there are discernible distances between BEX and BIN, as well as VEX and VIN, suggesting a practicality of this approach. It must be noted that this plot does not allow for conclusions to be drawn in terms of overlap. On average, it appears that the distances between vulnerable paths and centroids from different defect classes excerpt greater distances than their benign counterparts. However, it must be stressed that this also largely depends on the individual case.

Ultimately, it is also necessary to store the models in some way. Computationally wise, the calculation of the geometric centers is neglectable. Meanwhile, as was established in Section 13.2.1, the adaptability of the models is an important asset. For these reasons, it was thus decided that it would be best, to persist the models in terms of lists of paths, i.e., real-valued vectors. This allows changes to the models to be made by simply appending new paths to these lists. Vice versa, unwanted paths can be easily removed from the models if necessary. The models were then persisted as simple text files in the comma separated values (CSV) format. Processing the entirety of the Juliet test suite resulted in a very large model. Using the built in ZIP file library of the Python programming language, it was possible, however, to compress the file at a ratio of about 95%.

While the specific implementation of this demonstration pipeline focuses solely on the ICFG, it is possible to use other data structures as well, with manageable effort. In this situation, all that is necessary is to configure the SVF-phase accordingly and to adapt the queries to the desired graph. Next, a model would have to be built using the same query, for instance, by using the Juliet test suite for training data and finally, it might be necessary to make small adaptations to the last, distance-phase in the pipeline. Aside from the ICFG, SVF has the ability to generate SVFGs and Call Graphs. Further additions that were incorporated into the tool also allow for the extraction of the LLVM-IR code in graph form. None of these structures have been elaborated in depth, such that further research is necessary in order to evaluate their adequacy for ML-based SAST. Furthermore, as elaborated in Section 13.2, the prototype was designed in a framework-like fashion that allows for individual analysis phases to be exchangeable. This allows for the graph-generation step to also be exchanged entirely for some other tool that emits graphs. This way the prototype could also target other programming languages, such as the Java programming language.

13.2.6 Front End

As elaborated, one of the key strengths of this demonstration pipeline is its high degree of adaptability. This choice was made early on during the development process, as the lack of suitable training datasets for supervised machine learning approaches was deemed a critical and, as for today, not entirely remediable issue. Therefore, as a mitigation strategy, a feedback-loop was devised that, together with the high adaptability of the approach, would allow for the model to be improved upon using past findings. This feedback-loop is part of the demonstration pipeline's front end. As mentioned in Chapter 13.2.5.2, the models of the demonstration pipeline are simple lists of ICFG paths that are indicative of some underlying problem. The representation of these paths in a metric vector space is done one the fly, each time the prototype is run. Therefore, making amendments to these models is conceivably simple, as all that has to be done is appending new paths from new findings to this list. Figure 50 provides an overview of the steps involved in this process.

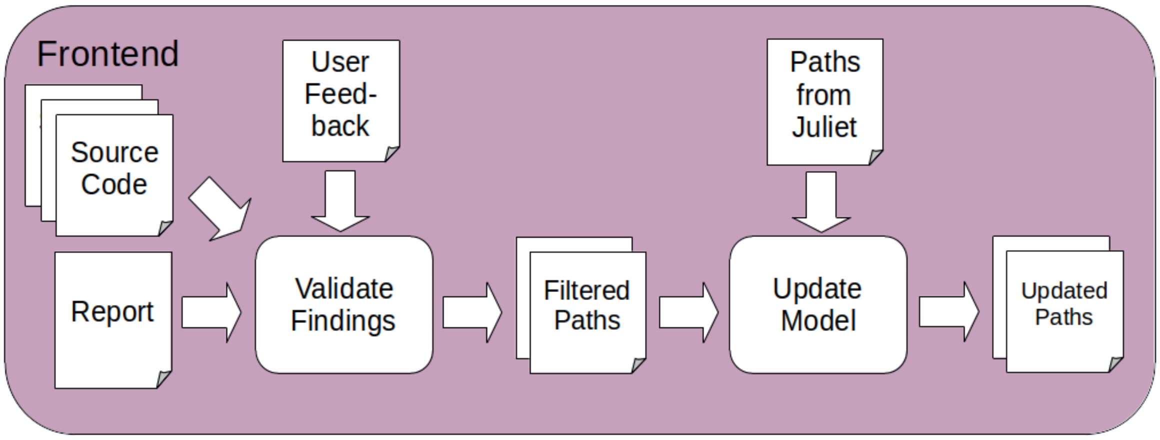


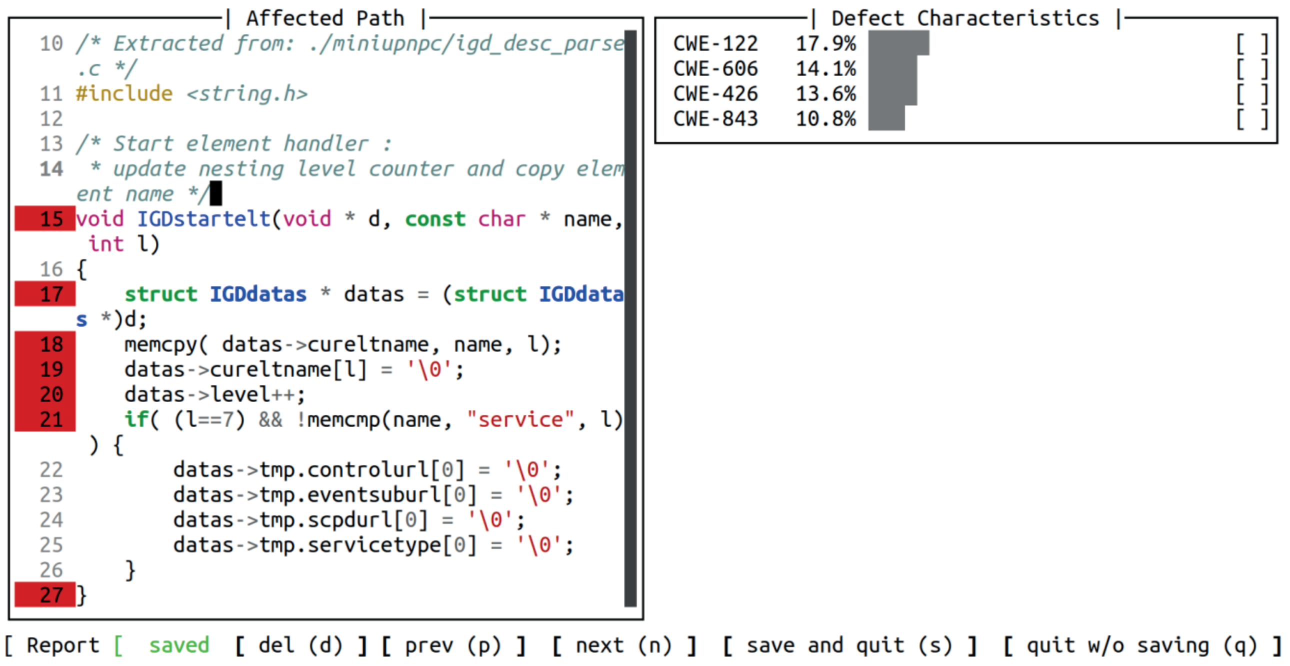
Figure 50: Schematic overview of the frontend.

The front end achieves this by not only presenting the results of the analysis, but also allowing the users to select individual findings (depicted by the *Filtered Paths* element in the above Figure 50) and the models that they wish to make the amendments to. Each finding of the demonstration pipeline is presented as a page, where the affected path is shown to the left. However, since the paths are internally represented as simple lists of ICFG nodes, they first have to be translated back into its program code representation. This process is rather straightforward, since the programs under analysis are compiled with the debug flag enabled and the source locations yielded this way, are stored in each and every intraprocedural ICFG node. Then, mapping the nodes back to their program code representation is as easy as retrieving this information for every node and loading the respective lines of code from the original source files. Since single lines alone are not very lucid by themselves, a delta is used that not only retrieves the affected line, but also a user-defined number of lines before and after. Moreover, as the ICFG is used, the paths extracted may cover multiple procedures, meaning that the context may switch repeatedly. So as to allow the users to keep track of which procedure they are viewing, every time the context changes, i.e., by means of a call or return node, the name of the current procedure is added to the source code in the form of a comment.

Another design choice with respect to the front end was that instead of reporting possible defects in terms of absolute findings, for each model the code was checked against, the distance to the centroid is expressed as a percentage. That percentage is to be read as the degree to which the finding corresponds to one of the models, i.e., in this case CWE classes. Not only does this allow users to prioritize and categorize findings, it is also a deliberate step away from viewing the problem of detecting defects in source code as a binary-decision problem. This takes advantage of the inherit principles by which the prototype detects defects and may help to reduce user fatigue due to the inevitable presence of false positives. If users are faced with a high number of false positive findings, they might dismiss all further findings after a while, as they may not see any use in the tool. If, however, reports are made in terms of the degree to which their code fulfills the characteristics of some defect class, the users may be more inclined to actually investigate the cause of this overlap. Figure 51 depicts a screenshot of the demonstration pipeline's front end. The defect correctly detected here, is a buffer overflow in the MiniUPnP[24] software, which was filed under the CVE entry: CVE-

[24] http://miniupnp.free.fr/

2015-6031[25]. On the left-hand side, the affected source code is visible. The ICFG paths that triggered the finding is marked on by the line numbers, highlighted in red. On the right-hand side, the defect characteristics for each model the code was checked against, are listed in descending order. Listed are only those models, where the affected path triggered an alarm by falling below one of the model's threshold. As can be seen, the CWE-class 122 was correctly reported as the most plausible type of error. This is not always the case, however, such that users should always investigate the code from the angle of every possible flaw reported.



```
| Affected Path |
10 /* Extracted from: ./miniupnpc/igd_desc_parse
   .c */
11 #include <string.h>
12
13 /* Start element handler :
14  * update nesting level counter and copy elem
   ent name */
15 void IGDstartelt(void * d, const char * name,
    int l)
16 {
17     struct IGDdatas * datas = (struct IGDdata
   s *)d;
18     memcpy( datas->cureltname, name, l);
19     datas->cureltname[l] = '\0';
20     datas->level++;
21     if( (l==7) && !memcmp(name, "service", l)
    ) {
22         datas->tmp.controlurl[0] = '\0';
23         datas->tmp.eventsuburl[0] = '\0';
24         datas->tmp.scpdurl[0] = '\0';
25         datas->tmp.servicetype[0] = '\0';
26     }
27 }
```

```
| Defect Characteristics |
CWE-122   17.9%   [ ]
CWE-606   14.1%   [ ]
CWE-426   13.6%   [ ]
CWE-843   10.8%   [ ]
```

```
[ Report [  saved  [ del (d) ] [ prev (p) ]  [ next (n) ]  [ save and quit (s) ]  [ quit w/o saving (q) ]
```

Figure 51: A screenshot of the prototype's front end.

If a user finds that a defect that was reported by the prototype is indeed a true positive finding, they may select the respective models to which they wish to append the affected path through the defect characteristics display on the right-hand side. The controls of the frontend are shown below the code. They allow the users in addition to step through the list of findings, hide false positives and save the changes to the model. Finally, it must be noted that the user interface was heavily zoomed in for this screenshot, so as to improve the readability.

13.3 Evaluation

Evaluating the prototype, i.e., the demonstration pipeline, was done in two different experiments. Firstly, the Juliet Test Suite was used, in order to determine the prototype's predictive performance in terms of the full confusion matrix that was introduced in Section 8.2. This is possible since the Juliet test suite is of synthetic nature, meaning that each test case therein was carefully crafted, such that it is safe to assume that a case labeled benign is indeed benign. For datasets that were created from real-world software projects, this property does not necessarily hold true, as elaborated in Section 11.1. It may be possible to determine sections of vulnerable code within a software project, but determining that a section is safe, on the other hand, is not easily possible, at least not without considerable amounts of effort. Therefore, using real-world data, only two dimensions of the confusion matrix may be considered: True positives and false negatives. Using the Juliet test suite alone, however, has the drawback that due to its synthetic nature, the test cases may not reflect real vulnerabilities to the full extent. In addition, the Juliet test suite was used for the creation of the models employed within the demonstration pipeline, hence evaluations performed on this ground truth alone may mask the overfitting problems. To mitigate some of these effects, it was therefore decided that in addition to the evaluation based on the Juliet test suite, another evaluation should

[25] https://nvd.nist.gov/vuln/detail/CVE-2015-6031

be conducted that is based on real-world data. However, due to the aforementioned limitations, only true positives and false negatives would be tracked. To this end, four different software projects were used: The LibTiff[26] library, a software library for handling image files in the "Tagged Image File Format" (TIFF), OpenJPEG[27], another library used to process images, but this time in the "Joint Photographic Expert Group" (JPEG) file format, MiniUPnP, an implementation of both, a client and a server, of the "Universal Plug and Play" (UPnP) protocol and finally the ZLib[28] software library comprising a number of data compression algorithms. All of the software projects are written in either the C or C++ programming language and should cover various types of software defects. In the ensuing sections, the experiments conducted on this basis will be dubbed as oracle tests, due to their characteristic of only allowing for yes- and no-answers on whether the prototype found a known vulnerability or not.

13.3.1 Juliet Test Suite Experiments

As has been elaborated during the introduction of this section, the Juliet test suite offers the benefit that it is known with certainty whether or not a particular test case is, indeed, benign. Therefore, the complete confusion matrix can be determined, which allows for more complex and expressive metrics to be calculated on the basis of these results. These metrics are not created equally and some that are often found in the literature are ill-applied and do not reflect the predictive performance of some tools to an adequate degree. A very obvious metric of this category would be the ratio between the correctly classified samples $TP + TN$ and the total samples $TP + FP + TN + FN$. This metric is referred to as the accuracy. Especially in the domain of SAST it should be avoided, however, as in cases of highly imbalanced classes, e.g., the natural distribution of vulnerable versus benign code samples, it tends to be overly optimistic in favor of the dominating class [131]. Nonetheless, this metric is often cited in other literature w.r.t. ML-SAST, hence it was decided to include it in the evaluation. Aside from the accuracy, the also very popular, but less problematic, precision, recall and F1-Score have been calculated during the evaluation. Section 8 covers the topic of metrics in much greater detail.

Since the Juliet test suite has also been used in the generation of the models used, it cannot be simply employed for the evaluation as is. This is due to the fact that using the same data a classifier was trained on, would not yield trustworthy and realistic results. Instead, the dataset must be split into a training and a test dataset, where particular caution must be exercised to not mix samples between the sets. Moreover, so as to avoid the possibility that the results yielded can be attributed to the particular split, the K-fold cross validation method is used. In this instance, $K = 10$ was chosen, as it can be considered common practice. With this method, the dataset is not split into two sets for training and testing, but ten. Then, one of the sets is reserved for testing, while the others are used for training. This is repeated ten times, where for each iteration another test set is chosen. The most evident method of doing so would likely be to simply process the entire test suite at once, leaving just the extracted ICFG paths and then making the split. This would be problematic though, as some of the paths in the test and training set could then originate from the same function, skewing the results of the experiment. Consequently, it is necessary to perform the splits at the very bottom of the pipeline, i.e., in the test suite itself. This is rather uncomplicated, fortunately, since the Juliet test suite is already organized in individual test cases, where there are no dependencies between them. That is, for each CWE class there is a directory in the test suite that contains a varying number of C or C++ files. These files are mostly self-contained and pose a single test case. The process of generating the datasets was thus designed as follows:

For each CWE class and each file therein, i.e., a test case, the entry points were collected and stored in list, which was then shuffled. Then, the list was split into ten equally sized sets and new main-procedures were generated that would call the entry points from these sets. The main procedure meant for testing would be comprised of a single fold, whereas the one for training would be generated from the remaining nine folds. This was done for all ten possible folds. Each main procedure resides in its own file, such that in a

[26] http://www.libtiff.org/
[27] https://www.openjpeg.org/
[28] https://www.zlib.net/

consecutive step, Makefiles could be generated that emit a single binary for each, the training and the test dataset. Figure 52 shows the procedure schematically.

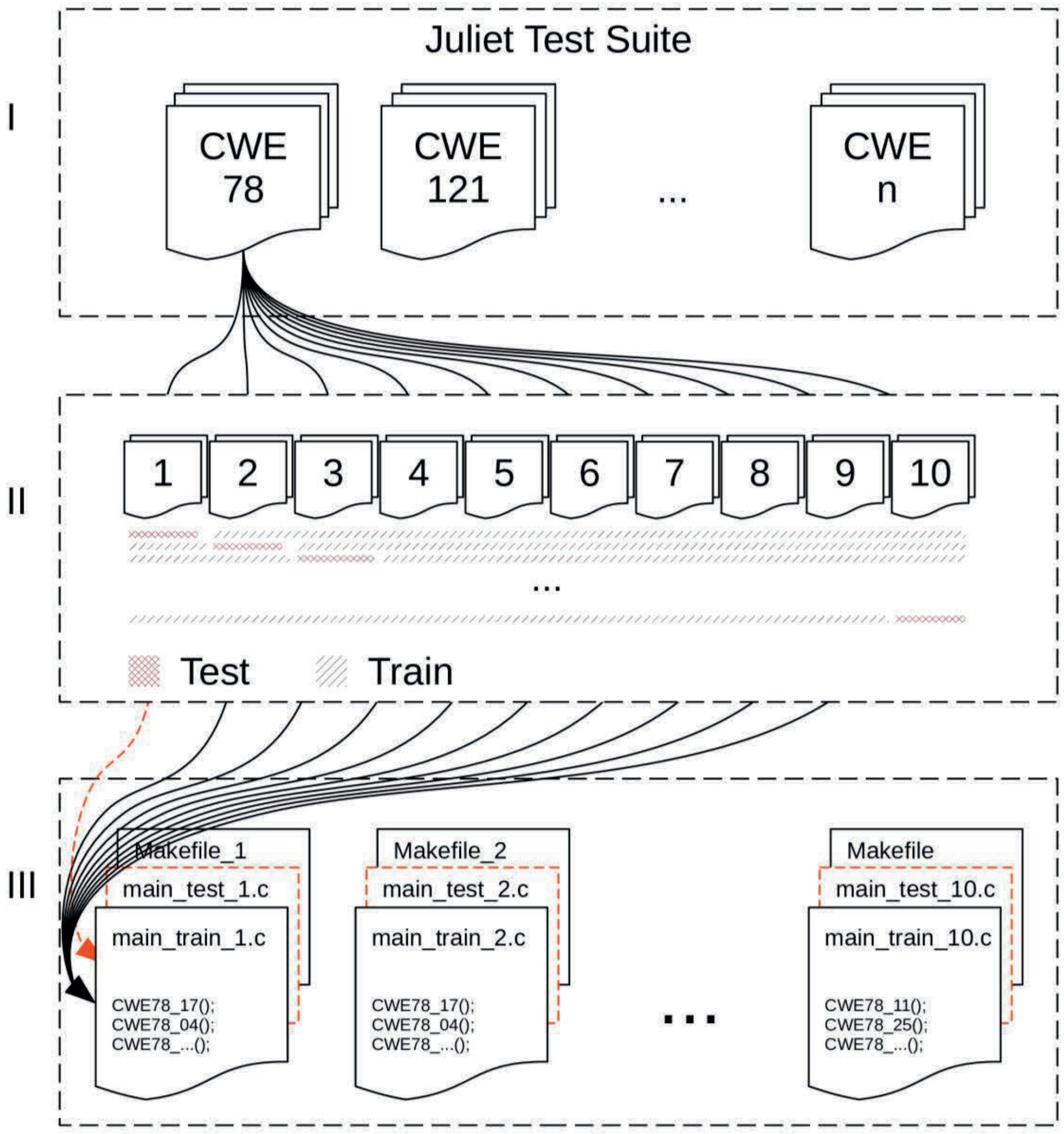


Figure 52: Schema of the procedure applied for the ten-fold cross validation using the Juliet test suite.

Here, beginning from the top, for each CWE class (I) the test cases are divided into equally sized chunks (II) and the splits are then formed by using nine of the ten chunks for the training and one for the test set, before being rotated (indicated by the hatched lines below). Then, on the example of the first split, the two main-procedures are created and the associated Makefile for each split (III). Afterwards the process continues as per usual and the Makefiles are used to compile and extract the LLVM-IR bit code from the train and test binaries. Then, the ICFGs are generated using SVF and finally the paths extracted by means of the Neo4j Database. The compilation and analysis of the Juliet test suite alone is by itself already a computationally exhausting task. Due to the fact that the ten-fold cross validation method multiplies this problem by a factor of ten, it took five days on an Intel E5-2690 V4 CPU with fourteen cores clocked at 2.60 GHz. It must be noted, however, that this process could have indeed been massively accelerated, had the task been parallelized more aggressively.

With the metrics and procedures described that are used in order to evaluate the prototype's predictive performance, the other aspect that is left to define is the method of grading the results. This problem is more involved than it may appear at first glance. This is due to the way that the prototype reports its findings. Originally the Juliet test suite was designed with conventional SAST tool in mind that report defective code on the level of lines and even columns. The prototype devised here, is only capable of analyzing paths in the interprocedural control flow, meaning that reports are made on a per-path basis. In the worst-case scenario, where each node on a path corresponds to an instruction in the LLVM-IR representation, a single path could span up to 75 nodes, as determined by the database query that is used, see 13.2.4. The actual number of lines is much smaller in reality, however, as a single line of code in the C or C++ programming languages usually corresponds to several instructions in its LLVM-IR representation. Furthermore, the maximum path length of 75 nodes is likely to be seldomly reached. Still the reports made by the prototype can and often do span multiple lines, therefore being incompatible with the manifest provided by the Juliet test suite. To alleviate this problem, it was thus decided that reports would be graded as is described in the following. Since the Juliet test suite uses a predictable pattern by which the procedures are named that pose the entry point for each test case, it is possible to tell benign from defective cases by this information alone. Furthermore, it is then possible to tell benign from defective paths, as they are extracted starting at these entry points. Using this information, it is possible to grade the results of the prototype on the same granularity, where for each path that the test suite reported as defective was considered a true positive (TP) if it was extracted from a defective procedure. If the path was extracted from an indeed benign procedure, the finding was booked as a false positive (FP). Vice versa, if the prototype failed to reported a path that was extracted from a defective procedure as such, the finding was deemed a false negative and finally, if the path was extracted from a benign procedure and also reported as benign, the finding was classified as a true negative (TN). Table 9 summarizes the rules applied for all four dimensions of the confusion matrix in a more concise fashion.

Table 9: The rules applied in order to determine the prototype's predictive performance in terms of TP, FP, FN and TN, i.e., the complete confusion matrix.

Dimension	***Description***
TP	A true positive was counted when the prototype reported an ICFG path that was extracted from a defective procedure.
FP	A finding was deemed a false positive, if the path analyzed was extracted from a benign procedure, but classified as defective by the prototype.
FN	A false negative was counted, if the tool did not pick up on a defect that is known to be extracted from a defective procedure.
TN	A true negative was attributed for every other path that the prototype did not report that was extracted from a benign procedure.

With the principles of the grading and the tenfold cross validation method laid out, one last thing to be discussed are the testcases themselves. Due to the high number of cases in the Juliet test suite and time restrictions it was not possible to test the prototype against every single case. Analyzing the entire Juliet test suite alone, already poses a computationally challenging, albeit manageable effort. The tenfold cross validation multiplies this effort by a factor of ten, however, which is out of scope for the time being. Hence, a selection of various defect classes was chosen, in hopes to paint a comprehensive picture of the prototype's predictive performance. The defects chosen range from more abstract types of errors, such as absolute path traversal vulnerabilities (CWE-36), command injection and format string vulnerabilities (CWE-78, CWE-134), buffer related defects (CWE-121 – CWE-127), numeric defects (CWE-190 – CWE-197), zero-division problems (CWE-369), resource mishandling issues (CWE-400 – CWE-590 and CWE-789) and unchecked return value defects (CWE-690). As for the metrics calculated from the results, the recall, precision, F1-score and the accuracy were chosen. The final results as well as the entirety of the CWE-classes the prototype was tested against are shown in Table 10. Furthermore, Figure 53 depicts the same results visually, by means of a bar plot.

Table 10: Results of the internal tenfold cross validation against select classes of the Juliet Test Suite.

CWE	***Recall***	***Precision***	***F1 Score***	***Accuracy***
36	0.816	0.766	0.790	0.686
78	0.260	0.912	0.405	0.272
121	0.393	0.778	0.522	0.483
122	0.345	0.789	0.480	0.426
124	0.374	0.810	0.512	0.431
126	0.430	0.860	0.573	0.468
127	0.434	0.797	0.562	0.466
134	0.468	0.979	0.633	0.483
190	0.741	0.965	0.838	0.752
191	0.736	0.965	0.835	0.748
194	0.332	0.602	0.428	0.402
195	0.375	0.604	0.463	0.390
197	0.236	0.654	0.347	0.292
369	0.781	1.000	0.877	0.783
400	0.632	0.978	0.768	0.646
401	0.953	0.994	0.973	0.955
415	0.831	0.958	0.890	0.881
590	0.567	0.788	0.659	0.596
690	0.855	0.905	0.879	0.895
789	0.666	0.997	0.794	0.661

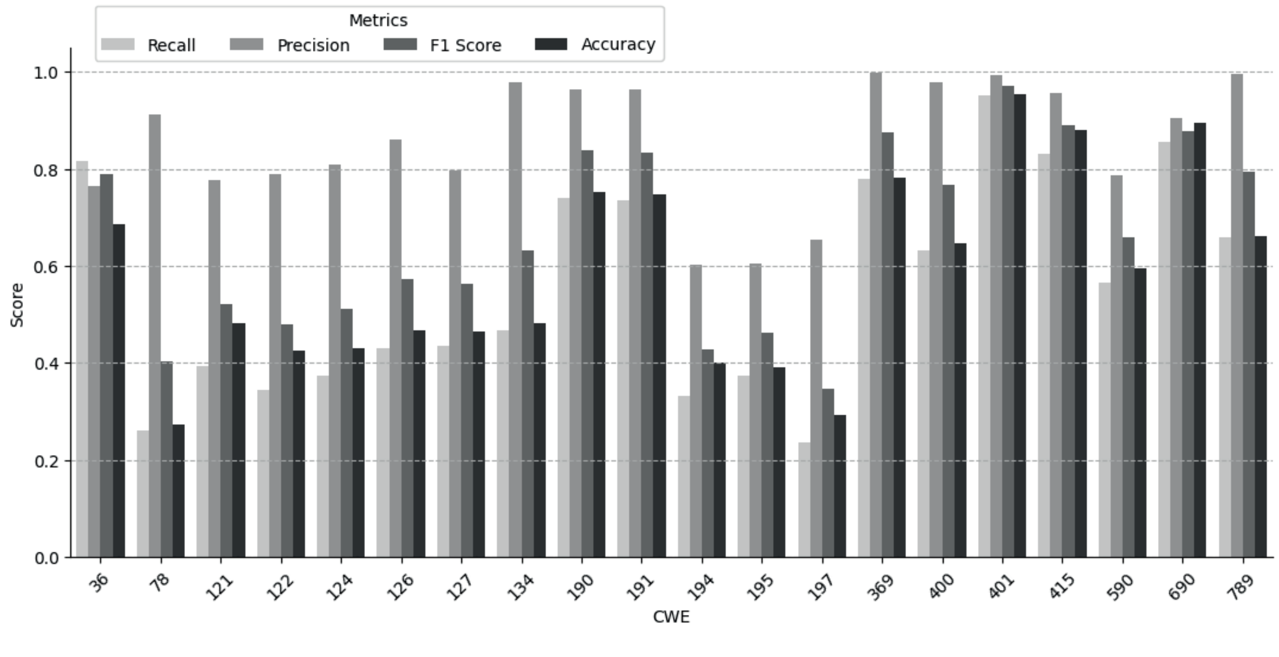


Figure 53: Categorical bar plot of results from the internal tenfold cross validation against select classes from the Juliet test suite.

As can be seen, the results of the evaluation vary greatly across the different types of software defects encountered within the Juliet test suite. While some types of defects appear to be detectable quite well, others are seemingly very hard for the prototype to pick up on. In particular, the numeric and buffer related issues are apparently challenging in that regard. Interestingly, however, out of the numeric type errors, the classes CWE-190 and CWE-191 stand with comparatively high scores across all metrics. This may, however, be explainable by the fact that these classes pertain to integer over- and underflow related defects, which, in contrast to the classes CWE194 – CWE-197, may be better reflected in the control flow. The better performing classes, which mainly comprise all starting at CWE-369 onwards, are also likely to be highly control flow related and thus, picked up by the prototype with ease.

13.3.2 Oracle Tests

As elaborated, the oracle tests were performed to gain insights into the real-world performance of the prototype and its demonstration pipeline. For this, a dataset by Fan et al. was used. The authors of the dataset crawled several open-source software projects' commit histories in order to determine commits that likely introduced some bug fix for a security related issue. The dataset also included references to the respective Common Vulnerabilities and Exposures (CVE) entries, as well as the according CWE class for the bugs [132]. From this dataset, a total of 50 different test cases across 44 different commits and four different software projects were selected, spanning 13 different CWE classes listed below in Table 11.

In order to conduct the analysis for each defect, a shell script was executed that cloned the software project, checked out the commit prior to the fix and ran the prototype on the source code in that version. For each defect, a report was generated by the prototype that contained paths, which were similar to paths of known defects. A Python script then filtered the reported paths for those that matched the filename and the line present in the defect. The findings for each defect were manually confirmed afterwards. This was necessary, as the findings might have been random, e.g., the prototype might have reported the correct location, but the detected defect characteristics would not match the underlying problem. If, for instance, the prototype reported a potential buffer overflow, but the actual underlying problem was a missing null-check, the report was dismissed as a false positive.

As elaborated in Section 13.2.5, either the opcode of the LLVM-IR instruction or the node's type may be used as embeddable features. For the oracle test, the former option was chosen, as the other academic tool that the prototype would be compared against, relied on this representation as well. The model, used to find similar defect characteristics, was initialized as outlined in 13.2.5.2, by randomly sampling 50% of paths extracted from the Juliet test suite to generate the Centroids and the other 50% to determine the optimal thresholds. Naturally, these had to be built from the same features, i.e., the opcodes. For each defect class, eight centroids were initialized on the defective paths. The scaling of the threshold to detect vulnerable paths was not altered and all paths that fell below the thresholds were added to the reports.

Out of all known defects of the oracle test cases, the prototype detected two. The prototype labelled them as being similar to characteristics of vulnerable paths from the defect class CWE 122. The commit messages belonging to the commits that fixed the vulnerabilities marked them as fixes for buffer-overflows. Combined with the manual inspection of the detected paths, these two defects were labelled as true positives. Furthermore, the prototype generated reports for seven out of the 44 known defects, which after manual inspection were labelled as false positives. They were labelled as such, because the defect classes that were reported did not match the defect mentioned in the git commit message or the defect which was estimated from inspection of the source code. Table 11, summarizes the findings of the prototype for the oracle tests. As mentioned before, two defects were correctly detected by the prototype (+). Due to the other seven findings not matching the correct defect characteristics, they were simply labelled as not found (-). The remaining defects were not detected and consequently booked as not found (-) as well.

To better understand how the prototype's performance compares to other academic as well as commercial tools, the same evaluation was conducted again, but this time using the FICS, as well as several commercial tools. FICS is an academic approach that was published by Ahmadi et al. under the Apache 2.0 License. The tool is a proof of concept and implements a technique, described in their 2021 paper "Finding Bugs Using Your Own Code". It employs clustering algorithms to find inconsistencies within the same codebase. To this end, the tool employs a two-step clustering technique based on the strongly-connected-components algorithm [133]. In the first step, clusters of code constructs are built that show some similarity. Then, within these clusters, smaller irregularities are detected, using a second, more precise clustering method. This way, functionally similar constructs are supposed to be found that are inconsistent in some detail. The idea is that such inconsistencies may be indicative of a software defect that could then be presented to an expert for further inspection [65]. As mentioned, aside from the FICS tool, several conventional SAST tools have been tested as well that are sold commercially. These tools must not be disclosed, however, due to their terms of use.

While FICS failed to analyze MiniUPnP entirely, it was at the same time able to detect three defects in the OpenJPEG software project that the prototype did not detect. The commercial tools, in contrast, failed to detect any of the defects. Table 11 shows the results of the oracle test in a concise fashion below.

Table 11: The results of the prototype, FICS as well as several commercial SAST tools on several defects is displayed. The columns F, P and C denote the respective tools: FICS, the prototype and the commercial tools, where each row shows whether the tool successfully discovered the vulnerability present in the previous commit (+) or not (-) or did not report anything at all (o). Note that the commercial tools have been aggregated into one single column, as neither of them detected any defect in the dataset.

CWE	Software Project	Commit Hash	F	P	C
CWE-119	miniupnp	79cca974a4c2ab1199786732a67ff6d898051b78	o	+	-
CWE-388	miniupnp	140ee8d2204b383279f854802b27bdb41c1d5d1a	o	-	-
CWE-125	miniupnp	b238cade9a173c6f751a34acf8ccff838a62aa47	o	-	-
CWE-119	miniupnp	7aeb624b44f86d335841242ff427433190e7168a	o	-	-
CWE-476	miniupnp	cb8a02af7a5677cf608e86d57ab04241cf34e24f	o	-	-
CWE-119	libtiff	ce6841d9e41d621ba23cf18b190ee6a23b2cc833	-	-	-
CWE-787	libtiff	5ad9d8016fbb60109302d558f7edb2cb2a3bb8e3	-	-	-
CWE-125	libtiff	ae9365db1b271b62b35ce018eac8799b1d5e8a53	-	-	-
CWE-119	libtiff	3ca657a8793dd011bf869695d72ad31c779c3cc1	-	+	-
CWE-119	libtiff	b18012dae552f85dcc5c57d3bf4e997a15b1cc1c	-	-	-
CWE-119	libtiff	5c080298d59efa53264d7248bbe3a04660db6ef7	-	-	-
CWE-119	libtiff	9657bbe3cdce4aaa90e07d50c1c70ae52da0ba6a	-	-	-
CWE-125	libtiff	9a72a69e035ee70ff5c41541c8c61cd97990d018	-	-	-
CWE-125	libtiff	1044b43637fa7f70fb19b93593777b78bd20da86	-	-	-
CWE-191	libtiff	5397a417e61258c69209904e652a1f409ec3b9df	-	-	-
CWE-369	libtiff	43bc256d8ae44b92d2734a3c5bc73957a4d7c1ec	-	-	-
CWE-369	libtiff	438274f938e046d33cb0e1230b41da32ffe223e1	-	-	-
CWE-189	libtiff	c7153361a4041260719b340f73f2f76	-	-	-
CWE-190	libtiff	787c0ee906430b772f33ca50b97b8b5ca070faec	-	-	-
CWE-787	libtiff	391e77fcd217e78b2c51342ac3ddb7100ecacdd2	-	-	-
CWE-369	libtiff	3c5eb8b1be544e41d2c336191bc4936300ad7543	-	-	-
CWE-119	libtiff	6a984bf7905c6621281588431f384e79d11a2e33	-	-	-
CWE-125	openjpeg	c16bc057ba3f125051c9966cf1f5b68a05681de4	-	-	-
CWE-119	openjpeg	e078172b1c3f98d2219c37076b238fb759c751ea	-	-	-
CWE-416	openjpeg	940100c28ae28931722290794889cf84a92c5f6f	-	-	-
CWE-787	openjpeg	dcac91b8c72f743bda7dbfa9032356bc8110098a	+	-	-
CWE-119	openjpeg	afb308b9ccbe129608c9205cf3bb39bbefad90b9	+	-	-
CWE-787	openjpeg	e5285319229a5d77bf316bb0d3a6cbd3cb8666d9	-	-	-
CWE-787	openjpeg	2cd30c2b06ce332dede81cccad8b334cde997281	-	-	-
CWE-119	openjpeg	baf0c1ad4572daa89caa3b12985bdd93530f0dd7	-	-	-
CWE-369	openjpeg	d27ccf01c68a31ad62b33d2dc1ba2bb1eeaafe7b	-	-	-
CWE-119	openjpeg	397f62c0a838e15d667ef50e27d5d011d2c79c04	-	-	-
CWE-119	openjpeg	162f6199c0cd3ec1c6c6dc65e41b2faab92b2d91	-	-	-

CWE	***Software Project***	***Commit Hash***	***F***	***P***	***C***
CWE-125	openjpeg	15f081c89650dccee4aa4ae66f614c3fdb268767	+	-	-
CWE-400	openjpeg	8ee335227bbcaf1614124046aa25e53d67b11ec3	-	-	-
CWE-190	openjpeg	c58df149900df862806d0e892859b41115875845	-	-	-
CWE-20	openjpeg	c277159986c80142180fbe5efb256bbf3bdf3edc	-	-	-
CWE-369	openjpeg	c5bd64ea146162967c29bd2af0cbb845ba3eaaaf	-	-	-
CWE-190	openjpeg	5d00b719f4b93b1445e6fb4c766b9a9883c57949	-	-	-
CWE-125	openjpeg	ef01f18dfc6780b776d0674ed3e7415c6ef54d24	-	-	-
CWE-189	zlib	d1d577490c15a0c6862473d7576352a9f18ef811	-	-	-
CWE-189	zlib	e54e1299404101a5a9d0cf5e45512b543967f958	-	-	-
CWE-189	zlib	9aaec95e82117c1cb0f9624264c3618fc380cecb	-	-	-
CWE-189	zlib	6a043145ca6e9c55184013841a67b2fef87e44c0	-	-	-

In light of these results, it is necessary to discuss the limitations of this evaluation, especially with respect to the commercial tools. Firstly, these tools were run, using their default configuration and rulesets. This is good in a sense that it allows for some neutrality, since manual configuration efforts might introduce some bias in favor of one or the other tool. However, another reason also is that the configuration of these tools is not trivial, due to the high complexity of them. It is very well possible that a trained practitioner, who is familiar with these commercial tools, would have gotten much better results in this test. Secondly, the dataset used for the oracle test was generated from existing open-source software projects that have been under development for decades in some cases. During this time, these software projects have been potentially forked and hosted at different sites, such that their past is not easily traceable. Hence, it is not known if and what kind of SAST tools their code base has been subjected to prior to the oracle test. This could likely be the case, which means that the defects, present in this dataset, are the ones that are extremely hard to detect for conventional SAST tools. For one thing, this means that the evaluation puts the commercial tool at an unfair disadvantage. At the same time, however, this would show the benefit of ML-based SAST tools, which is not that they are necessarily better than existing solutions, but instead being potentially capable of detecting different defects that the conventional tools failed to detect. Hence, ML-based SAST tools could make up for a valuable addition to the toolbox of existing SAST solutions.

In this context, it is also worthwhile to take a closer look at the FICS tool in comparison to the prototype. As can be seen, this approach was able to detect three defects from a different software project that the prototype did not detect. Vice versa, the prototype unveiled two defects that FICS did not detect. This may also highlight that since both tools work entirely different from each other, FICS detects project intrinsic defects, whereas the prototype detects more general defects, it could be beneficial to use both combined. Furthermore, it must be stressed that the prototype, so far, only relies on the artificial Juliet test suite as a dataset for the model generation. Incorporating paths from real-world projects into the existing model, for example through the feedback loop that comes with the prototype, might help to increase its performance on the defects in the oracle tests. Moreover, with better datasets, gathered using the built-in feedback loop, the prototype's performance may be increased further. Lastly, for now the prototype only uses the ICFG alone, in order to detect software defects. This data structure lends itself to the detection of control-flow-based issues, such as missing return value checks. Using other data structures, such as the SVFG, it may be possible to detect other types of defects as well that are better modeled using the data flow. Problems that, for example, manifest in the data flow could likely be modeled better using this data structure. Lastly the used embedding, a basic mapping of the OP-code onto numerical values, might have been too simple to form a generalized representation of the underlying defects.

13.4 Limitations and Threats to Validity

Beginning with the presuppositions from the literature study, it has already been known that the application of machine learning techniques in the research area of SAST is associated with some difficulties. To date it seems that a lot of these difficulties remain hard to overcome. Namely, these comprise a) a lack of suitable training and therefore also evaluation datasets, and b) data structures that are adequate for the task or methods that would allow to faithfully embed graphs into a vector space. In this section, these limitations will be explored and their impact on the validity of this, and possibly also in extension to other ML-SAST approaches, discussed.

Training- as well as test-datasets are an important factor in the learning and evaluation phases of ML-based SAST tools. As was the case for this very prototype, the lack of such datasets meant that the efficacy of the approach could only be determined up until a certain point. As elaborated, there are two different types of datasets available: Synthetic sets that were manually crafted and real-world datasets that comprise samples from actual software projects. Beginning with the latter, the absence of knowledge about the true negatives is most difficult in this context. While it is easy to prove that some program is defective, as showing the existence of only one single defect in that program is necessary to do so, it is much harder to prove that a program is free from defects. In that case, it is necessary to prove that over the entirety of all possible input combinations, the program performs in accordance with its specification. Realistically, this feat is practically impossible, as already stated by Rice's theorem [1]. This is why during the oracle test in Section 13.3.2, no attempts were made to measure the true negatives and false positives. To do so, one would have to validate that except for the defect in question, the rest of the software under analysis is free from defects. This, of course, is impossible.

Due to this circumstance, it is hard to determine the actual practicality of this approach in realistic scenarios. Synthetic datasets, on the other hand, are not ideal either. The Juliet test suite, used for the statistical evaluation in Section 13.3.1, does offer the benefit of being synthetic in nature. This means that it is safe to assume that all code labeled benign, is indeed, most likely benign. Hence, using this dataset, it is possible to also make statements about the false negative and true positive dimensions of the confusion matrix. On the other hand, this also means that the samples contained in it are potentially less realistic and may not represent real software defects to their full extent. Moreover, the Juliet test suite was not only used for the evaluation of the prototype, but also for the training of its models. While great care was taken to prevent data snooping by mixing train and test samples using tenfold cross validation, the data between the train and test sets may ultimately still be rather similar. This may hold particularly true for the Juliet test suite, due to its homogeneous and repetitive nature. The results of this evaluation therefore show that the described approach appears to be intrinsically valid, being able to distinguish between different program behaviors with respect to the control flow. The significance of these results in terms of real-world predictive performance, on the other hand, might be compromised.

When considering the aspect of model training in particular, these circumstances may have a negative impact on the generalizability of the models. Whereas in many cases ill-chosen sample sizes are to blame for this overfitting of models, in the case of the prototype and in extension other ML-based SAST approaches, different reasons might exist to blame. The idea of overfitting often implies that there are indeed some general patterns in the data, which the model failed to pick up on. Considering programs the question arises if there even is such thing as a general defect pattern that some model could pick up on. Arguably, computer programs are fundamentally different than the types of data that are found in other research areas that employ machine learning techniques. Being very complex in nature, alterations in program behavior may be very subtle and nuanced. The same code that is considered benign in one part of a program, may pose a dangerous defect in another part of the same program. Then, the difference between both parts lies solely in their context, something that is easily missed depending on the window the program is observed through. Unfortunately, this window is very small for machine learning algorithms. This circumstance is further exacerbated by the fact that although being an amenable data structure in the context of conventional static analysis, when paired with machine learning, graphs are very difficult to embed into a vector space.

These properties of computer programs may give rise to a very low signal-to-noise ratio when attempting to separate benign from defective code. It remains to be seen whether this signal-to-noise ratio can be improved in the future or if there even are general patterns that underlie some concrete examples of software defects. Proving or disproving the existence of these general patterns, however, will remain a tough endeavor, if even possible. As a starting point, a key requirement for this would be to improve the available datasets. The prototype aims to contribute to this answer in so far as it heavily lends itself to the idea of easily and rapidly retrainable models, in hopes to accelerate this development. As for now, solving these problems on a basic level may not be possible.

On a more technical level, there are some other limitations pertaining to the extraction mechanism used to gather the ICFGs paths. Firstly, due to the nature of this mechanism of extracting every unique path within a procedure, from the entry node to the exit node, the total number of paths that need to be traversed may grow exponentially in the worst-case scenario. This would be the case if the procedure under analysis is solely comprised of branch instructions. Since the maximum depth of the path traversals is limited to 75 nodes, a total of 2^{73} distinct paths would have to be traversed (the entry and exit nodes need not be counted). This would render the approach impractical, however, this being an extreme edge case, it is unlikely that this kind of scenario would be encountered in real world applications. During all tests conducted on real software projects, including those listed in Section 13.3.2, no such problems could be observed.

Another limitation regarding the extraction method is that there might be some small chance of introducing label noise. As explained above, the maximum depth the extracted paths may reach is limited to 75 nodes. If the defect happens beyond this point, it would not be part of the extracted path. Hence a path that is labeled as defective might actually be benign in such scenario. The model used here that was built upon the Juliet test suite, only ever reached 75 nodes in about one percent of all cases and even then, it is not certain that the defect happens outside of these paths. Another scenario where paths may wrongfully be labeled as defective, could occur when some defect does not interact with a particular execution path at all. It is unknown to what degree this circumstance excerpts an influence on the predictive performance of the model. The intrinsic evaluation in Section 13.3.1 seems to suggest that the model is very well capable of discerning between defective and benign paths. However, some filtering mechanism that isolates the defective behavior to a better degree, might be able to improve the predictive performance further.

Finally, the approach chosen is very simplistic in nature to allow for rapid retraining of the models. Not only for this reason, but also for the explainability of the models, this is favorable. On the downside of this, however, is the fact that other more complex models, such as neural networks, may offer a drastically improved predictive performance when compared to this approach. It might be argued in defense of the prototype that the data used is more important than the choice of machine learning algorithm. If the data is useless, so is the model trained on this basis.

13.5 Summary and Discussion

This chapter covered the implementation of an ML-based SAST approach, describing the development process and specifically the challenges encountered. Based on the interviews described in chapters 5 and 6 and the literature studied in Chapter 10, it was determined that instead of solely focusing on the predictive performance, other and in particular more fundamental aspects require attention more urgently. Due to the statements made during the expert interviews, a lack of suitable test- and training datasets, as well as concerns regarding the overall generalizability of ML-based SAST, it was decided that the adaptability of the models used, is the area that the attention should lie on. Therefore, a light-weight analysis was implemented that focuses on control-flow-based software defects and that can be quickly adapted to new training data, once available. The contributions of this approach are hence the following:

Firstly, due to the ability to rapidly retrain the models, the users are enabled to configure the tool to the specific needs of each individual software project. Not only may this mitigate the effects of poorly generalizing models due to unsuitable training datasets, but it also offers benefits in terms of the tool's usability. Whereas conventional SAST tools require users to write their own rulesets for the adjustment of the tool to their specific needs, the approach outlined here is purely example based. This means that users do not need to define explicit rules, but mere examples of the wanted or unwanted program behavior suffice to make the necessary adaptations. Moreover, it is hoped that due to these properties, this approach may help to generate more suitable datasets in the future.

With respect to the literature studied it must also be noted that there appears to be a gap between the actual current state of the art and the methods described there. In particular, it seems that the fundamental difficulties in embedding programs into a vector space and the lack of suitable training and test datasets to the end of model learning are underrepresented in the literature. Furthermore, the extraction of program data, significant for the detection of vulnerabilities, remains equally an important, yet often overlooked, factor in that regard. Due to the small size of input data the machine learning algorithms can process at a time, this is particularly difficult, but important for the models to generalize well from concrete software defects. Due to the aforementioned limitations in training and test data, it is hard to quantify these correlations, unfortunately. In order for ML-based SAST approaches to succeed, solving these fundamental challenges is detrimental.

On an even more basic level, one could also raise the question whether there even is such a thing as general defect patterns that underly concrete software defects. In the literature, this basic prerequisite is often times simply assumed, i.e., there are some common patterns that multiple vulnerabilities share that are invisible to the human observer. Conveying this sentiment may be considered dangerous, however, as it suggests a false sense of completeness regarding ML-based SAST approaches. Users of such tools may then overestimate their capabilities, being lulled into a false sense of security. Ultimately, it is impossible to give a definitive answer to the question of their existence, but assuming some intermediate scenario, where some limited form of generalizability is provided, it may be better to set realistic expectations in terms of ML-bases SAST tools. As is the case with conventional SAST, it may be as important for ML-SAST to adapt to the specific characteristics of different software projects instead of attempting to provide a silver bullet that is meant to work across all of them. In some cases, where it is possible to generalize from specific behavior, models may of course be provided, but their limitations should be made transparent to the user. These aspects should be the topic of further research, to allow for a better assessment of the applicability of ML-based SAST in realistic scenarios.

Given these circumstances, it may also be possible to argue that it is not the models which require adjustment, but instead the general expectations towards machine learning based SAST approaches. Lowered expectations in terms of the generalizability of such do not equate to a complete disregard of their applicability. When extending the application of ML to static analysis techniques in general, similar approaches would possibly do well in the field of clone detection, which in itself could also be used in the context of SAST.

The generalizability of the approach chosen for the prototype is also tightly linked to the data structures used for the analysis. This subject is hard to evaluate systematically, since there are many variables to be considered and each of these may have a drastic influence on the overall performance of the approach. First of all, a program representation must be chosen. In this case, the ICFG appeared to offer an adequate signal-to-noise ratio that allowed for distinguishable patterns between defective and benign code to become visible in the data, i.e., the Juliet test suite. Aside from the ICFG that was used, it is certainly possible to use other program representations for the analysis. Another interesting choice in that regard would be the call graph of the programs, which could not be experimented with, due to a limited time budget. After the graphs have been built, they must be decomposed into units, which are processable by machine learning algorithms. As for the prototype, single paths were chosen, due to problems regarding methods to embed more complex graphs. Then there is also the question of what data to embed, as there are multiple options available, even in the case of the comparatively simple ICFG. Other program representations, such as the SDG or SVFG, comprise even more and in particular more complex data, e.g., points-to-sets that contain references to abstract objects in memory. The prototype simply uses the node types from the ICFG, as they appeared to offer a good signal-to-noise ratio in the experiments, due to their small alphabet of only six different types, when also marking split and merge points in the paths. As can be seen, the signal-to-noise ratio of the extracted data is pivotal to successfully achieving models that generalize well. At this point in time, this remains a challenging task, since choosing data that is too specific appears to lead to the models failing to represent the underlying patterns adequately. If the data is too generic, on the other hand, the models may not be able to effectively distinguish between defective and benign behavior, leading to a high rate of false positives.

Using synthetic datasets, in return, such as the Juliet test suite appears to suffer from the detriment of not being very realistic and hence skewing the models in a way that they do not generalize very well. The prototype described here aims to mitigate this effect to some degree, by allowing for the models to be quickly adapted to other code bases and possibly more realistic samples, should these eventually become available. This is a trait that common supervised approaches do not share with the prototype, as they require laborious retraining procedures. Ultimately, clustering approaches may pose a more elegant approach to ML-SAST in general, not requiring labeled datasets at all. As elaborated earlier in this chapter, this indeed was the original approach the experimentation started with. However, during that time it became evident that this approach, while offering the benefit of not requiring labeled training data, ultimately suffers from other disadvantages.

14 Conclusion

This study comprehensively outlined recent developments in regards to the application of machine learning techniques for static application security testing (ML-SAST). Furthermore, based on the results of the literature study, a prototype has been developed under a free and open-source license. The process and the thereby gathered experiences have been described in detail in Chapter 13. As for the literature study, three different sources of information have been tapped, to gather insight into the requirements demanded by actual practitioners as well as the current state of research regarding ML-SAST:

1. Interviews with software security experts were conducted, in order to gather their opinions and expectations towards ML-based SAST approaches and SAST in general.
2. The opinions of SAST-tool users concerning the application of machine learning for static application security testing were inquired through an online questionnaire.
3. Following the guidelines suggested by Petersen et al., a systematic mapping study, employing the snowball-sampling method recommended by Wohlin was conducted[91], [92].

The latter of the three contributions yielded a total of 72 highly relevant publications on the topic of ML-SAST. Moreover, these findings were thoroughly discussed by means of extensive quantitative as well as qualitative analysis, elucidating promising approaches in the research area of ML-SAST and highlighting the challenges yet to be solved.

Based on the techniques and problems described in the literature, the prototype that has been devised employs supervised, as well as clustering methods to distinguish between defective and benign code. Aside from the properties and challenges already known from the literature study, the development of this prototype helped us uncover further problems. Therefore, as a way of mitigating some of the uncovered problems, the following provisions have been made: Firstly, since no suitable embedding method for the interprocedural control flow graph could be made out during the development, the prototype uses a novel technique that uses individual execution paths in order to map the program code into a metric space. When evaluated using the Juliet test suite, it could be shown that this kind of representation lead to clearly discernible program behavior in the vector space. Furthermore, the lack of suitable training data was addressed by incorporating a feedback loop into the prototype, which allows users to refine models on the basis of previous findings. This was made possible due to the lightweight and adaptable nature of this approach and its models. The resulting tool is planned to be released to the public under an open-source license.

In summary, ML-SAST appears to be an emerging field of research with an ever-increasing relevance as reflected in the expert interviews as well as the literature identified during the course of this study. As of today, there still remain unanswered questions, subject to further research.

14.1 Further Research

As this study has revealed, there are still gaps in the area of ML-SAST that highlight the potential for further research. These mainly comprise a lack of suitable training data for supervised approaches, embedding methods for complex program graph representations and finally, a lack of free and open-source frameworks for static analyses. The prototype, devised in this study, aims so mitigate these problems to some degree, but still further research is necessary in these areas as outlined in the following section.

Starting with the training data sets, most of those available tend to be either insufficiently realistic or suffer from a skewed ground truth. The impact of this assumption is twofold, since such data sets are not only essential for training, but also for evaluation purposes. Firstly, the synthetic datasets, such as the Juliet test suite, are likely not very realistic and may not depict the true complexity of real software defects to the fullest degree. Realistic data sets on the other hand, may suffer from a skewed ground truth and in addition, a lack of context. The prototype devised in this document aims to mitigate some of the associated problems, by introducing a feedback loop that allows for the reintroduction of previous findings into the models. Ideally, however, this problem should be tackled at the root. Aside from the training itself, such datasets are also necessary for the evaluation of the models. While it was shown that the prototype has the capability of effectively distinguishing between different program behaviors, the evaluation to this end was mostly conducted on test data from the same dataset the models were initially trained on. This does ensure intrinsic validity; however, the external validity could only be established to a limited degree, due to the lack of sound and realistic evaluation data sets, see Section 13.3.2. As it stands, this problem seems to be rarely discussed in the literature. Unfortunately, the creation of sound and realistic data sets for the training of ML-SAST approaches is an extremely challenging task. Thus, it may only be addressed through extensive community effort and the potential help of commercial vendors of static analysis tools. The latter, in particular, may offer access to much more insightful data than bug trackers or version control systems of open-source software can provide. In order for ML-SAST to succeed in the future, these problems require urgent discussion.

On a more basic level, early on during the development of the prototype it was also shown that there are few free and open-source tools being developed for the static analysis of programs, written in the C or C++ programming languages. These tools are important, as they are required for determining the data and control dependencies within a program and for points-to analyses. Especially noticeable in that regard was a lack of suitable code slicers for these programming languages. Any additions to the repertoire of tools or the continued development and improvement of existing tools are thus a welcome development. In general, it seems that a lot of the publications yielded during the literature study appear to neglect conventional methods that the static analysis has to offer. Instead, it looks as if the focus lies entirely on the use of more capable machine learning algorithms in a lot of cases. However, the surrounding circumstances may be slightly more complex than apparent at first glance: One of the more popular tools in the research area of ML-SAST, appears to be the Joern tool, often used in the preprocessing of the source code. It is only now that the authors of the tool, seem to address the development of a points-to analysis for the tool, which in turn means that all previous ML-SAST approaches using Joern, do not perform such analysis at all[29]. Further research is needed in order to determine the impact that not providing machine learning algorithms with this kind of information during training has on the resulting models. Depending on the outcome, appropriate measures should be taken.

Further down the pipeline, it was also discovered that graph embedding methods are equally scarce and, at least in the context of ML-SAST, unsuitable for now. As elaborated in Section 13.1.2 and Section 13.2.4, program graph representations are complex heterogeneous graphs that pose a challenge to the already rather limited graph embedding methods. Most of these methods either pertain to the topology of the graphs alone or the content of the nodes, but seldomly both. Therefore, future research might attend to either improving these methods in general or at least adapting these methods to be better suited for program graph representations. Alternatively, the energy might also be focused on finding alternative program representations that are better suited for ML-based SAST applications.

As for the prototype in particular, it was found that most problems are best modeled by using the unwanted behavior, i.e., known defective ICFG paths, and searching for similar patterns in previously unseen code. The problem can also be inverted, however, if the centroids are formed from benign paths, where a higher distance is considered to be unfavorable. In a few isolated cases, better results were yielded this way. In those cases, the common ground appeared to be the fact that the vulnerable code showed a higher degree of similarity than the benign code. Thus, it is easier to define the bad behavior as it forms tighter clusters in the vector space. Ultimately, it is recommended to use the defective centroids, to model the unwanted behavior, but users of the prototype are invited to experiment with both approaches. In what scenarios each of the approaches is appropriate, is a topic for further research. In that context it was also found that mixing paths from different defect classes to generate the centroids from, lead to decreased predictive performance, as mentioned in Section 13.2.5.2. This could be caused by paths from different classes of

[29] https://github.com/joernio/joern/pull/1249

defects, being widely scattered in the vector space. Hence, they are hard to cluster and the resulting centroids may not be very representative of the underlying problems. The exact circumstances and possibles remedies were not explored during this study, due to time constraints. It may be worthwhile to focus further research efforts on this, as understanding this circumstance may lead to other improvements, with respect to this method.

Additionally, one direction in which the research could be continued is the extension of the tool in terms of other programming languages. At least for the Java programming language a version of the Juliet test suite exists and, in fact, a much broader selection of suitable analysis tools, to generate the necessary program graphs with. With respect to the graphs, it would be also interesting to see how other types of program graph representations could be employed to model other types of vulnerabilities. Even with SVF as the basis of the prototype, the possibilities that the available SVFG and PTACG open up, are yet to be explored. Using the SVFG, for instance, it would be possible to model defects that manifest in the flow of data, rather than control. The PTACG, on the other hand, could potentially allow for defects to be modelled more explicitly that are caused by procedures being called in the wrong order or not being called at all. However, the ICFG already includes the PTACG implicitly, so that it would be recommendable to start with the SVFG in terms of further research opportunities. Furthermore, it needs to be established yet, what kinds of defects exactly are well detectable using the approach outlined in this document. Future research may attempt to further define these boundaries and to find applications in which the approach could be combined with other existing SAST approaches, to find a beneficial symbiosis.

Finally, on a larger scale, it may then also be necessary to determine in general, what kinds of problems ML-based SAST approaches are good at detecting. Such insights may be helpful in differentiating ML-SAST from conventional SAST and find suitable application scenarios for this technology. Another question in this context is also, what kind of ML-algorithms are even suitable for ML-SAST. It may then also be worth investigating, what kind of impact the choice of algorithm has on the resulting model's predictive performance, in contrast to the soundness of the training data or the associated static analysis techniques applied during the preprocessing phase. Current trends seem to suggest that graph neural networks are gaining popularity. However, this type of neural network is held back by a lack of adequate explanation methods [111]. If there existed better explanation methods for graph neural networks, they would be more widely adopted for the task of ML-based SAST. Using graph neural networks for ML-based SAST would additionally come with the benefit that no explicit graph embedding methods are necessary, which would circumvent the problems that pertain to the embedding of program graphs.

14.2 Outlook

As outlined, there are still a lot of connecting points for further research in the area of ML-based SAST approaches. While the application of machine learning methods to the problem of SAST is likely not going to refute the undecidability of a program's semantics, it may be very well capable of finding solutions for individual instances [1]. Therefore, machine learning for SAST poses an interesting field of research, however, there are still a lot of problems that the community should address. With respect to this, key findings of this study include:

1. A distinct lack of adequate datasets, that hinder the development of more capable models.
2. Graph representations of programs remain difficult to embed in a vector space, calling for research of more suitable embedding methods.
3. Free and open-source tools for static analysis purposes are scarce but as important for this emerging field of research to prosper.
4. Graph neural networks seem to offer increased predictive performance in comparison to other supervised learning methods, but are held back by their immature explanation methods.

Generally, there have been a multitude of different approaches to ML-SAST, most of which with a high focus on improving on some selected metrics. Comparisons, however, still remain difficult due to the lack of suitable datasets as a common baseline. In light of this observation, the creation of adequate datasets for the sake of training, testing and comparing different approaches should be considered paramount.

Moreover, while there indeed appears to be room for further improvements on raw vulnerability detection performance, a topic largely overshadowed by this is the usability of machine learning-based SAST approaches. This aspect, however, should not be disregarded. The interviews with experts, as well as the extensive search for literature on ML-SAST, have shown that practitioners regard to SAST tools as generally useful, but that it would be necessary to define project-specific detection rules. The definition of such rules, may pose a challenge to some, which is where possibly the greatest opportunity for ML-based SAST lies. Being an example-driven approach, users are not obliged to conceive their own rules, but merely need to be able to tell benign from defective program code apart.

15 Appendix

15.1 Glossary

Term	*Description*
2-norm	The 2-norm, also called Euclidian norm, is the length or magnitude of a vector.
Abstract Syntax Tree (AST)	A tree-like data structure that is usually build during compiling source code and which store information on statements, variables, etc. of a program. It can be used for certain static program analyses.
Accuracy	Is a metric used to determine the predictive performance of a classifier. It is defined as $\mathrm{Acc} = \frac{\mathrm{TP}+\mathrm{TN}}{\mathrm{TP}+\mathrm{FP}+\mathrm{TN}+\mathrm{FN}}$ and thus only weighs in the correct predictions made. This limits the metric's expressiveness in situations of high class imbalance, such as is often the case in SAST applications. See also See also *True Positive Rate (TPR)*, *True Negative Rate (TNR)*, *False Positive Rate (FNR)*, *False Negative Rate (FNR)* and *Confusion Matrix*.
Anchor Questions	Questions that are used in interviews, to guide the conversation back to the main topic.
Area Under the Curve (AUC)	The area under the curve refers to the area enclosed by the receiver operator curve (ROC) and is indicative of a classifiers discriminative power. A perfect classifier scores an AUC of 1, whereas in theory the worst-case scenario would be a classifier that scores an AUC of 0, i.e., always predicting the wrong class. A Classifier scoring 0.5 as it's AUC would practically even worse however as it's discriminative performance would be on par with randomly guessing.
Array Boundary Errors	Refer to a type of software defect where a memory outside of the area defined by an array data structure is accessed.
Automatic Garbage Collection (GC)	A method employed by some programming languages for the automatic management of memory. Such implementations for instance track the number of references to any object in memory and marks them for deletion once that counter reaches zero.
Backward Snowballing	A scientific method used in systematic literature studies to identify further, unknown literature by examination of the works referenced in known literature on some topic. The method was first introduced by Claes Wohlin [92]. See also forward snowballing and snowball sampling.
Bag of Words (BOW)	A representation method used in machine learning to generate features from text. The occurrence of each word - from a common vocabulary - in a text is counted, producing a numerical vector for each text. This vector can then be used in machine learning. Transforming the sentence "I like ice-cream" would produce the vector: [1, 1, 1, 0] while transforming the sentence "I like pizza" would result in vector: [1, 1, 0, 1]

Term	***Description***
Banach's Fixed Point Theorem	Is a fixed-point theorem that guarantees the existence, uniqueness and of a fixed point for a given function. It can be used to show the convergence of a fixed-point iteration. In case of GNNs it is used to prove the existence of a solution wrt. the state of the network.
Bidirectional Gated Recurrent Unit (BGRU)	The bidirectional version of gated recurrent unit type neural networks, see *Gated Recurrent Unit (GRU)*.
Black-Box Explanation Method	A method for the explanation of predictions made by machine learning models. Unlike its counterpart, the white-box explanation method, such methods require no knowledge about any parameters of the network. Explanations are only inferred by observation of changes in the output of a model caused by careful alterations of some input. See *LEMNA*, *SHAP*, *Explanation Method*, *Explainable AI (XAI)* and *White-Box Explanation Method*.
Buffer Overflow	A type of software defect that occurs when data is written beyond the boundaries of the allocated area in memory. Buffer overflows are a common cause of software vulnerabilities and may be used as an entry point for the execution of malicious code by an attacker.
Call Graph (CG)	A graph data structure that stores the call dependencies of functions/methods of a program. It is, for example, used for certain static program analyses.
Chain-based Clustering Methods	Discovery of clusters with arbitrary shape. The prominent clustering algorithm DBSCAN by Ester Kriegel, Sander and Xu is an example of such density-based notion of clusters with one parameter for the user to determine an appropriate value [134].
Class Imbalance	Is a term used in the context of machine learning to describe the circumstance when there exists a class within the data that dominates all others in number by a large margin.
Clustering	A fundamental approach to machine learning that aims to group unlabeled data in clusters, by inferring hidden patterns.
COCO train2017 dataset	A large dataset of annotated images for the purpose of training machine learning models. See: https://cocodataset.org/
Code Property Graph (CPG)	A graph-based representation of source code, first introduced by Yamaguchi et al. in 2014 [36]. It combines abstract syntax trees, control flow and program dependence graphs in a singular data structure.
Common Vulnerabilities and Exposures List (CVE)	A public tracking system for known vulnerabilities in software. vulnerabilities submitted are classified and rated according to their type and severity. See CWE.
Common Weakness Enumeration (CWE)	A taxonomic view of software vulnerabilities that is constantly revised. The vulnerabilities are organized in a hierarchical order that groups them based on their properties.

Term	***Description***
Communication	Refers to a a qualitative requirement for applications involving artificial intelligence that defines rules as to how the users of such techniques are informed about their limitations, capabilities and even the fact they are interacting with AI, e.g., AI may not pretend to be a human agent. See also *Transparency*, *Explainibility* and *Traceability*.
Concurrency problems	A class of software vulnerabilities that relates to defects caused by the concurrent execution of program logic, for instance by conflicting accesses to a shared data structure.
Confidence	A measure used in machine learning to quantify the certainty of a prediction from a machine learning model.
Confusion Matrix	A form of representing the four basic metrics TPR, TNR, FPR and FNR in a matrix. See also *True Positive Rate (TPR)*, *True Negative Rate (TNR)*, *False Positive Rate (FNR)* and *False Negative Rate (FNR)*.
Control Flow Graph	A data structure used by static program analyses that represents the control flow of the program (i.e., execution order of the statements).
Convolutional Neural Network (CNN)	A family of neural networks that lends its name from the convolution layers present. These layers work by moving a kernel over the input, which detects important features and generates an activation map for the consecutive layers. Convolution layers are often followed by some pooling layer, to down sample their output. At the end of such networks there is usually a number of fully connected layers.
Correlation Matrix	Is a 2D table to show the pairwise correlations of features and is used, e.g., in Kalman filters. In singular value decomposition it also allows to rank features and cut off the ones less influential.
Cross-Validation	Describes a technique often employed during model training so as to assess its generalizability and select the best performing settings for a model. A popular choice is k-fold cross-validation. Here the dataset is evenly split into k partitions. Then $k-1$ partitions are used for training, the last is reserved for testing. This way all possible permutations are exhausted.
Dangling Pointers	Describe pointer variables that refer to areas in memory that have already been released and possibly reallocated in a different context. Accessing these locations in memory leads to undefined program behavior and may entail serious security implications. See also *Use-After-Free*.
Data Provenance	Data provenance refers to all meta data, i.e., artefacts that accrue during the creation process of the data itself. In terms of machine learning this may refer to the datasets used during model training, but also the information on how these datasets were conceived. Data provenance is an integral part in supporting the requirement of traceability and ultimately transparency.

Term	***Description***
Decomposition Methods	Are methods employed to infer post-hoc explanations for the predictions of machine learning models that cannot be otherwise interpreted. In order to determine important features of a models input, its output is traced back to the input layer by decomposing the model. Deep taylor decomposition is a rather bold example of such methods.
Deep Belief Networks	Are a type of highly connected deep neural network architecture, introduced by Hinton et al.
Deep Representation Learning	The application of deep learning techniques for the automatic inference of relevant features in data samples.
Defect Prediction	Refers to the act of predicting the likelihood of defects in software, based for instance on code metrics or more complex analysis of the control or data flow of a program.
Descriptive Accuracy	Is a concept used in the context of explanation methods for machine learning models. It refers to the accuracy of the explanations yielded by such methods, e.g., in the domain of ML-SAST it may be of interest to gain insight into what tokens of the code supplied to the model had a high impact on the predicted class. Then, the accuracy would be the number of tokens whose impact has been correctly assessed divided by the total number of tokens in the sample [57]. See *Accuracy*, *Confusion Matrix* and *Explanation Methods*.
Descriptive Sparsity	Is a concept used in the context of explanation methods for machine learning models. It measures how sparse the explanations given by such methods are, e.g., in case of ML-SAST it is desirable to know which tokens had an impact on the predicted class and only those that had an impact. An explanation method that fails to discriminate between tokens with a low or a high influence, would be considered of low descriptive sparsity and is thus of little use to the practitioner [57]. See also *Graph Sparsity*.
Devign Dataset	Refers to the dataset used during training of the ML-SAST approach proposed by Zhou et al., named Devign [56]. The dataset is partially available to the public, with the remainder seemingly withheld by the authors for unknown reasons.
Distance Measure	A measure to summarize the similarity or dissimilarity of two objects.
Doc2Vec	Describes a learning based method for the generation of numerical embeddings of text documents.
Draper V-Disc	A dataset for Ml-SAST applications, consisting of samples mined from GitHub repositories that was labeled using conventional SAST tools [46], [84].
Dynamic Application Security Testing (DAST)	A technique complimentary to Static Application Security Testing (SAST) that needs a running instance of the program under analysis.
Dynamic Time Warp	An algorithm to measure similarities between temporal sequences which may vary in speed.

Term	*Description*
Editing Distance	The number of operations required to transform a given sequence into another sequence, e.g., the editing distance between “time” and “lime” could be considered as requiring a single replacement of the letter “t” with “l”. The distance between “time” and “thames” would comprise two insertions and one deletion, resulting in a distance of three, by the same metric.
Error Use Case	Source code examples that cannot be detected with the help of SAST tools.
Explainability	Is a criterion referring to the possibility to give explanations, as to how the decisions made by artificial intelligence came to be. Many so-called black-box models do not offer that possibility and their decisions are thus opaque to the practitioners using them.
Explainable Artificial Intelligence (XAI)	Refers to approaches to artificial intelligence that are explainable. See *Explainability*.
False Negative Rate (FNR)	The False Negative Rate is one of the basic metrics for evaluating the predictive performance of machine learning models that can be found in the confusion matrix. It is the number of samples predicted as negative despite being in actuality positive.
False Positive Rate (FPR)	The False Positive Rate is one of the basic metrics for evaluating the predictive performance of machine learning models. It is found in the confusion matrix and represents the number of samples predicted as positive despite being in actuality negative.
Fault Prediction	The act of predicting the likelihood of defects occurring in a given software unit. To this end metrics, pattern matching or as this study aims to explore machine learning techniques may be employed.
F-Measure	Also referred to as F-Score or $\mathrm{F_1}$-Score, is a more complex metric often used for the evaluation of classifiers. It is defined as the harmonic mean of precision and recall: $\mathrm{F_1} = \frac{\mathrm{TP}}{\mathrm{TP}+0.5(\mathrm{FP}+\mathrm{FN})}$
Format String	Refer to character string templates that comprise place holder variables to be dynamically filled in during execution and can be found in many different programming languages, e.g., C or C++. Format strings may be the cause of software vulnerabilities if for instance user input is not properly validated, allowing adversarial users to inject unexpected place holder variables that when evaluated lead to undefined behavior.
Forward Snowballing	A method used in systematic literature mappings to identify previously unknown literature by studying works that refer to literature, already known. The technique was first introduced by Claes Wohlin [92].

Term	***Description***
Gated Recurrent Unit (GRU)	Refers to the hidden activation function used in GRU type neural networks. They may be considered an advanced form of LSTM networks in a sense that they use fewer gates and abandon the concept of cell state, making them easier to implement and train.
General Data Protection Regulation (GDPR)	Is a regulation in the EU for data protection and privacy. One of its main intentions is the protection of personal data.
Gradient Descent (stochastic)	Refers to a technique that is used for finding local minima of a function, based on the function's gradient. It is applied to the loss function during the training phase of neural networks, to find out how to adjust the networks parameters, so as to minimize the loss. See also: *Loss Function*, *Backpropagation* and *Neural Network*.
Graph Classification	Refers to the classification of whole graphs according to some scheme, as opposed to *node classification* that is only concerned with the classification of individual vertices. See *node classification*.
Graph Neural Networks (GNN)	Are a type of neural network that are specifically tailored towards processing graph-like data. Graph neural networks adapt to the topology of the input data, thus efficiently incorporating the information contained therein.
Graph Sparsity	A metric used to assess the capability of explanation methods for graph neural networks, to clearly distinguish between relevant and irrelevant nodes for a given prediction [111]. See also *Descriptive Sparsity*.
Heartbleed Bug	A serious vulnerability in the OpenSSL library that affected the so-called heartbeat extension of the TLS and DTLS implementations. It is filed under CVE-2014-160 and allowed remote attackers to access unintended locations in main memory, thus allowing the extraction of possibly confidential information.
HDF5-Format	A hierarchical file format for storing large quantities of scientific data that is also commonly used in the context of machine learning, e.g., for the interchange of training and test datasets.
Hyper Parameters	Are parameters that are not adjusted by the algorithm, but rather set prior during conception. Hyper parameters may for instance refer to the dimensions of the layers, the number of total layers or the batch size during training of a supervised deep neural network.
Informedness	Is a metric used for the evaluation of binary classifiers and given by $\text{IFN} = \frac{\text{TP}}{\text{TP+FP}} - \frac{\text{FP}}{\text{TN+FN}}$, with TP, TN being the number of samples correctly predicted as either positive or negative and FP, FN being the number of erroneously predicted as either positive or negative. For some applications this metric is arguably more expressive than simpler ones, e.g., Accuracy. It considers both positive and negative samples, important when the population of samples exhibits high class imbalance. See also *Markedness* and *Confusion Matrix*.

Term	*Description*
Initialization Vector (IV)	A cryptographic primitive that refers to a nonce value required by some ciphers such as for initialization purposes. In case of the Advanced Encryption Standard (AES) in its (256-Bit) Counter Block Chaining (CBC) mode for instance, a suitable IV would be a 256-Bit long random value that forms the first cipher text input of the chain.
Integer Over- / Underflows	Are a type of software defect that occurs when a integer is incremented or decremented beyond the minimum or maximum value it can hold. A 32-Bit integer for instance, may not be incremented beyond 4294967296 or decremented below 0 before rolling over. An attacker with the ability to cause an over- or underflow may exploit this behavior to access unintended areas of memory.
Integrated- / Gradients	Is a white-box algorithm for explaining predictions made by MLPs, CNNs and RNNs. The simple gradients-based method so-called saliency maps, given by $\mathrm{r_i} = \frac{\partial \mathrm{y}}{\partial \mathrm{x_i}}$, i.e., how much y changes wrt. some $\mathrm{x_i}$. It was first introduced by Simonyan et al. and later revised by Sundararajan et al. [57]. See also *White-Box Methods, Black-Box Methods* and *LRP*.
Interpretability	Is a concept that is sometimes used interchangeably with explainability. However, in some works of literature a distinction is made, where interpretability is a higher criterion. It postulates that the model itself is already able to provide explanations, so that no additional methods to this end are required. Decision trees are often attributed to the class of interpretable AI. See also *Explainability* and *Explainable AI*.
Interprocedural Static Analysis	A static program analysis that considers several functions along a call path.
Intraprocedural Static Analysis	A static program analysis that only considers one function or method (cf. Interprocedural Static Analysis).
Json File Format	A human- as well as machine-readable data exchange format. See also YAML File Format.
Juliet	A dataset comprising synthetic examples of software defects for benchmarking purposes, maintained by the National Institute of Standards and Technology.
Kalman Filtering	Probabilistic method for robots to increase the certainty via iterative update on state variables due to observation. Often linear models are used to represent noise in the actions and perception, but for extended filters non-linear functions can also be used.
Kernel Trick	Is a technique used for the separation of data samples in kernel-based machine learning algorithms. Data that cannot be efficiently separated in a low dimensional vector space, may become distinguishable in higher dimensional spaces. A popular example based on this method, are support vector machines (SVM).
K-Means	An algorithm to partition a set of observations into k groups. It tries to minimize the sum of squared distances of each datapoint to its respective group center.

Term	***Description***
K-Nearest Neighbor	A classification algorithm which classifies each datapoint according to its k nearest neighbors.
Layer-Wise Relevance Propagation (LRP)	Is a white-box explanation method for a variety of artificial neural network models. Given a prediction $\hat{y}$ for some input vector x, this method backpropagates the state of the output layer through the network, such that the relevance score at the input layer can be calculated, i.e., the influence of each individual dimension of x excerpted on $\hat{y}$. Furthermore, the LRP method preserves the conservation property for each layer of the network [135]. See also *Explanation Methods*, *White-Box Explanation Methods*, *Integrated- / Gradients*.
Leave-One-Out Evaluation	Is the extreme case for X-fold cross valuation with a training vs. test set ratio X:1.
LEMNA	A black-box explanation method proposed by Guo et al. intended to be used for security related applications. It is based on a mixed regression model, i.e., it calculates the weighted sum of K linear models to approximate an unknown model [57].
Levenshtein Distance	A type of distance metric that is used to measure the editing distance between two sequences based on the operations: Insert, delete and replace.
LIME	A perturbation-based black-box explanation method that aims to approximate an unknown decision function $\mathrm{F_N(x), x \in \mathbb{R}^n}$ by randomly zeroing-out parts of x. The resulting vector $\tilde{x}$ is then used to make a prediction $\mathrm{F_N}(\tilde{x}) = \hat{y}$, so as to approximate the local Neighborhood of the function at point $\mathrm{F_N(x)}$. The decision boundary is approximated using a weighted linear regression model [57]. See also *SHAP* and *Black-Box Explanation Methods*.
Logical Errors	This term refers to a certain kind of software security flaw that manifest in the security
Logistic Regression	The disadvantage of isotonic regression is that it creates a lookup table for converting scores into estimated probabilities. An alternative is to use a parametric model. The most common model is called univariate logistic regression.
Loss Function	Also referred to as cost or error function is a mathematical function used in the learning process of neural networks and similar machine learning techniques. The function associates a cost to a prediction made by the model, usually by means of measuring the distance between the prediction and the actual target.
Malware	Adversarial software showcasing malicious behavior.
Markedness	Is a metric used for the evaluation of binary classifiers and given by $\frac{\mathrm{TP}}{\mathrm{TP+FP}} - \frac{\mathrm{FP}}{\mathrm{TN+FN}}$
Matthews Correlation Coefficient (MCC)	A complex evaluation metric used to assess the predictive performance of machine learning models for binary classification tasks. Unlike many other metrics, the MMC remains sound even in the face of severe class imbalance, such as is often the case with machine learning-based approaches to SAST.

Term	***Description***
Memory Leak	A type of software defect common in programming languages that require manual memory management. If memory is allocated but erroneously never again released during program execution, the process or even operating system may run out of memory.
MEX Vocabulary	Is the name of a lightweight schema for machine learning experiments that is based on the W3Cs PROV Ontology [89]. See also *PROV-O* and *W3C*.
Multilayer Perceptron (MLP)	One of the earlier and simpler types of deep neural networks. Multilayer perceptrons are comprised of at least three layers of fully connected nodes and thus commonly considered a “deep” architecture, as some of the layers are not directly observable.
Needleman-Wunsch	Is an multiple sequence alignment algorithm used for approximate string matching. The main difference to the Smith-Waterman algorithm (see below) is that negative scoring matrix cells are set to zero, which renders the (thus positively scoring) local alignments visible.
Negative Predictive Value (NPV)	A metric used in the assessment of the predictive performance in classification tasks, see *Recall*.
Node Classification	Refers to the task of predicting the affiliation of a graph’s nodes to some class. See also *Graph classification.*
Null Pointer Access	Describes a type of software defect that occurs in the C and C++ programming languages, when a pointer to the invalid memory address 0x0 is dereferenced. Other programming languages have similar concepts, but may differ in their implementation, e.g. None in Python 3 is a special instance of the None-class.
Overfitting	The concept overfitting means that a model is fit too good on its training data. Thus leading to a good performance on the training data while having a bad performance on unseen data.
OpenSSL Software	Refers to a cryptographic library and software suite that hosts a large variety of algorithm and protocol implementations for securing data transmissions over computer networks, such as the commonly used Transport Layer Security Protocol (TLS).
Over- / Under-Sampling	Refers to a technique used in the context of machine learning, so as to combat class imbalance in datasets. While under-sampling refers to the removal of data samples from the dominating class, over-sampling aims to restore an equilibrium by (artificially) increasing the number of data samples in the minority class. A popular example is the Synthetic Minority Over-sampling Technique (SMOTE).
OWASP	Is the commonly used abbreviation to refer the non-profit organization “Open Web Application Security Project”. The organization aims to improve software security [136]
Perturbation Methods	Are a family of black box explanation methods that aim to infer information about the local neighborhood of an unknown prediction function at a given point through careful manipulation of the input vector and measurement of the effect it has on the prediction.

Term	*Description*
PKL File Format	A file format that is used for the serialization of objects in the Python programming language. It can be commonly found in the context of machine learning applications, as it is used to preserve models, datasets and other related artifacts.
P-Norm	A vector norm belonging to the family of p-norms if $\|\|x\|\|_p = (\sum_{i=1..n} \|x\|^p)^{(1/p)}$, for any p > 0. Prominent are the (p=1)-norm $\|\|x\|\|_1 = \|x_1\| + \|x_2\| + \dots + \|x_n\|$, the (p=2)-norm or Euclidean norm $\|\|x\|\|2 = \mathrm{sqrt}(\|x_1\|^2 + \dots + \|x_n\|^2)$ and the (p=∞) norm $\|\|x\|\|_\infty = \max_{i=1..n}\|x_i\|$.
Positive Predictive Value (PPV)	A metric used for evaluation purposes in classification tasks, see *Precision*.
Precision	In some cases also referred to as "Positive Predictive Value" (PPV), is a metric often used in the assessment of the predictive performance of classifiers and the counterpart to the Recall metric (NPV). It can be directly calculated from the confusion matrix and is defined as $\mathrm{PPV} = \frac{\mathrm{TP}}{\mathrm{TP+FP}}$ where TP is the number of samples correctly predicted as belonging to the positive class and FP the number of samples erroneously predicted as belonging to the positive class. See also *Recall*, *NPV* and *Confusion Matrix*.
PROV Ontology	Is an ontology developed by the W3C that implements the PROV data model for data provenance, i.e., metadata relating to artifacts that arise from the creation of the original data.
Random Forests	A selection of decisions trees used in an ensemble to predict the classification outcome.
Rebalancing Methods	Are methods for the restoration of an equilibrium in imbalanced datasets. See Class Imbalance and Over- / Under-Sampling.
Recall	Sometimes called "Negative Predictive Value" (NPV) is the counterpart to the Precision metric (PPV). It is used for the assessment of the predictive performance of classifiers and given by $NPV = \frac{TN}{TN+FN}$ where TN is the number of samples correctly predicted negative and FN, the number of samples wrongfully predicted negative. See also *Precision*, *PPV* and *Confusion Matrix*.
Receiver Operating Characteristic (ROC) Curve	A type of plot used to characterize a classifiers ability to discriminate between classes, where TPR and FPR are plotted against each other, while shifting the discrimination threshold
Reinforcement Learning	Unlike supervised or unsupervised learning, where there exists a distinct training and deployment phase, reinforcement learning poses a machine learning paradigm were models are continuously trained. See also *True Positive Rate*, *False Positive Rate* and *Area Under the Curve*.
Reliability	High-stakes decision-making in areas like healthcare, finance and governance requires accountability for decisions and for how data is used in making decisions. Reliability in machine learning unifies questions of model stability, fairness and explanation.

Term	***Description***
ReVeal	A set of training data for the purpose of learning ML-SAST approaches. It contains a large number of real-world code samples in the C and C++ programming language, labeled as benign or vulnerable. It was curated by Chakraborty et al. and first introduced in their 2021 evaluation study on deep learning-based vulnerability detection approaches [137].
Robustness	A criterion to determine the robustness of explanation methods for machine learning models against perturbations, see
Rule-based SAST Solutions	Are approaches to static analysis security testing using a fixed set of rules, representative of generalized patterns of vulnerable source code. The source code is evaluated against these rules, subsequent pattern matches may indicate the presence of a software vulnerability.
SARD Project	The Software Assurance Reference Dataset (SARD) Project hosts a large repository of vulnerable code examples in a multitude of different programming languages.
SEI CERT C Coding Standard	A security-oriented coding standard for a variety of programming languages, such as C and C++ that is continuously revised by the Carnegie Mellon University's Software Engineering Institute.
Sensitivity	A basic metric found in the confusion matrix used for the evaluation of classifiers, see *True Positive Rate*.
SHAP	A perturbation based black-box explanation method proposed by Lundberg and Lee. It is based on the LIME method, but uses the so-called SHAP kernel as it's weighting function [57]. See also *LIME* and *Black-Box Explanation Method*.
Slicing Mechanism	A term first coined by Mark Weiser in 1981, originally referring to the act of human practitioners, syntactically abstracting from a program by reducing it to a minimal form while preserving the semantics [138]. It has nowadays come to mainly represent the automatic process of (automatically) extracting parts of a program at point p that comprises all statements that exercise some influence on a variable x at p, following the definition of Horwitz et al. from 1988 [37].
Smart Pointers	Smart pointers are used in many popular programming languages such as C++. These are special pointers that have additional functions and properties compared to simple pointer variables. To avoid accidentally releasing memory areas that are still referenced, reference-counting pointers are used. The smart pointer contains a counter variable that is incremented each time the pointer is copied. The memory is only released when the counter reaches the value 0.
Smith-Waterman	Approximate string-matching algorithm based on dynamic programming used in multiple sequence alignment. Relates to computing the edit distance.

Term	***Description***
SMOTE	Is the abbreviation of commonly used for the Synthetic Minority Over-sampling Technique, a method to artificially increase the number of samples belonging to the minority class in imbalanced datasets [98]. The method is meant to combat the negative effects of highly imbalanced training data machine learning models. See also *Over- / Undersampling*.
Snowball Sampling Method	Is a method for finding additional and previously unkown literature to some given topic based on a number of known works, first introduced by Claes Wohlin [92]. See Backward Snowball Sampling and Forward Snowball Sampling.
Specificity	A basic metric found in the confusion matrix used for the evaluation of classifiers, see *True Negative Rate*.
Stability	Is a criterion used in the context of explanation methods that are non-deterministic, to retain the same explanations across multiple runs, given the same input [111].
Static Application Security Testing (SAST)	A class of security testing tools usually working on the source code of the software under analysis. SAST is an important part of a Security Development Lifecycle.
Static Program Analysis	Program code is analyzed statically, mostly on the source-code level in contrast to dynamic analyses where the application under test has to be running.
Stream Ciphers	Describes ciphers that, in contrast to their block-wise counterparts, encrypt and respectively decrypt continuous streams of data.
String Termination Errors	Is a kind of software defect that may be present in programs, written in the C or C++ programming languages. Here strings are terminated by a null character. Such errors occur for instance if the presence of this character is not accounted for when determining the length of a string or reading in data in expectance of such null character, which may result in a buffer overflow. See Buffer Overflow.
Structural Robustness	Refers to the robustness of an explanation method to retain the same explanation, even when the input is perturbated. Ideally slight alterations of the input of such methods should except little effect on the explanation yielded [111].
Supervised Learning Process	A paradigm of machine learning algorithms, where there exists a distinct training and deployment phase in a model's lifecycle and training is accomplished by using labeled data samples, i.e., samples representative of the underlying problem and their associated labels, to be predicted.
Support Vector Machines	Refers to a supervised machine learning technique that aims to separate classes by construction of a hyperplane. Separation is achieved by constructing the hyperplane in a way that maximizes the distance between the plane and the nearest datapoints. As in most cases a linear separation is not possible in the original vector space, it must be lifted into a higher dimensional space using a kernel function, commonly referred to as the "kernel-trick".

Term	***Description***
Surrogate Methods	Refer to a class of black-box explanation methods that aim to mimic the behavior of an unknown und unexplainable machine learning model, by training another machine learning model that is explainable. See also *Explainability*, *Explainable AI* and *Black-Box Explanation Methods*.
Symmetric Key	A secret that is used in symmetric key ciphers for the encryption as well as decryption of data. Unlike its asymmetric counterpart, where there exist two separate keys for these tasks, with symmetric encryption both the sending and the receiving party of an encrypted message need to share the same key.
Systematic Mapping Study	A study of scientific literature that follows a systematic approach, so as to scope an academic research field.
System Dependence Graph (SDG)	Similar to program dependence graphs, system dependence graphs are a form of code representation that combines control and dataflow in a single graph. While the former encapsulates intraprocedural relationships, system dependence graphs work on an interprocedural level. See also *Interprocedural Program Analysis*, *Intraprocedural Program Analysis*, *Program Dependence Graph (PDG)*.
Tainted Data	Data that is controlled externally (e.g., by an attacker) and flows into certain program points. If tainted data reaches vulnerable code locations, often attacks are possible.
Test and Training Set	Are datasets containing a number of labeled samples that are used to train and evaluate supervised machine learning models. First the model is learned using the training set. Then the test set is used evaluate the performance of the model using different metrics. See also *Confusion Matrix*, *Supervised Learning*, *etc*.
Text Regular Expression Matching	The act of searching for substrings in text, by employment of regular grammars. For SAST is must be noted that these types of expressions, (usually) cannot encapsulate the full semantics of the programming languages they are trying to analyze, as they are weaker. Unlike these programming languages, the question whether a particular word lies within a given language is decidable however for regular expressions.
Traceability	Is a qualitative requirement regarding machine learning applications. It refers to the ability to trace back the origins of the data as well as meta data leading to the output of the model. See also *Data Provencance* and *Transparency*.
Transparency	Is a basic requirement concerning machine learning applications. It comprises explainability, traceability and communication. See also *Explainability*, *Traceability*, *Communication* and *Data Provenance*.
Transport Layer Security (TLS)	A protocol for cryptographically securing TCP connections so as to ensure the basic security goals of maintaining confidentiality and integrity of the data transmitted. In its current version 1.3, the protocol is standardized in the IETF Standard: RFC 8446.

Term	*Description*
True Negative Rate	One of the four basic metrics of the confusion matrix, also known as “Specificity” used in the evaluation of the predictive performance of classifiers. It is defined by the number of samples correctly predicted as negative. See also *True Positive Rate*, *False Positive Rate*, *False Negative Rate, Sensitivity, Specificity* and *Confusion Matrix*.
True Positive Rate	One of the four basic metrics of the confusion matrix, sometimes also referred to as “Sensitivity”, used in the evaluation of the predictive performance of classifiers. It is defined by the number of samples correctly predicted as positive. See also *True Negative Rate*, *False Positive Rate*, *False Negative Rate, Sensitivity, Specificity* and *Confusion Matrix*.
Use-after-free	Refers to a type of software defect that occurs when a pointer variable references an area in memory that has already been freed and has been possibly reallocated in a different context. Dereference of such pointer variables leads to undefined program behavior. See also *Dangling Pointers*.
Value Flow Graph (VFG)	A synonym of the term Data Flow Graph (DFG). See also *Data Flow Graph*.
Void Pointers	Pointer variables that, at least conceptually, have no datatype declared.
White-Box Explanation Method	Are a family of explanation methods used to gain insights into the prediction process of otherwise opaque machine learning models. This kind of explanation requires all (hyper-) parameters of the model to be known, unlike their black-box counterparts. See also, *Explainable AI (XAI)*, *Explanation Methods*, *LRP*, *Integrated- / Gradients* and *Black-Box Explanation Method.*
World Wide Web Consortium (W3C)	A consortium for the standardization and maintenance of web technologies.
XFG	Subgraphs, i.e., slices, of the program dependence graph. A term solely used in the DeepWukong paper by Cheng et al. [55]. See also *Program Slicing* and *Program Dependence Graph (PDG)*.
YAML file format	Is a human- and machine-readable interchange format. See also *JSON File Format*.

15.2 Results of the Literature Mapping

Listed here are the results of the snowball sampling, executed during the course of the literature Mapping, described in Chapter 10. The base set consists of 19 publications, while the first round yielded 15, the second round 15, the third round 21 and finally the fourth round a mere 4 results. In total, 72 publications have been sighted, which includes those from the base set.

15.2.1 Base Set

C. Batur Şahin and L. Abualigah, “A novel deep learning-based feature selection model for improving the static analysis of vulnerability detection,” Neural Comput & Applic, May 2021, doi: 10/gk52p3.

X. Cheng, H. Wang, J. Hua, G. Xu, and Y. Sui, “DeepWukong: statically detecting software vulnerabilities using deep graph neural network,” ACM Trans. Softw. Eng. Methodol., vol. 30, no. 3, p. 38:1-38:33, Apr. 2021, doi: 10/gk52pz.

R. Ferenc, P. Hegedűs, P. Gyimesi, G. Antal, D. Bán, and T. Gyimóthy, “Challenging machine learning algorithms in predicting vulnerable JavaScript functions,” in Proceedings of the 7th International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering, Montreal, Quebec, Canada, May 2019, pp. 8–14. doi: 10/ggzsh3.

F. Fischer et al., “Stack Overflow Considered Helpful! Deep Learning Security Nudges Towards Stronger Cryptography,” 2019, pp. 339–356. Accessed: May 26, 2021. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/fischer

S. M. Ghaffarian and H. R. Shahriari, “Software Vulnerability Analysis and Discovery Using Machine-Learning and Data-Mining Techniques: a survey,” ACM Comput. Surv., vol. 50, no. 4, p. 56:1-56:36, Aug. 2017, doi: 10/gftfv7.

J. Jiang, X. Yu, Y. Sun, and H. Zeng, “A Survey of the Software Vulnerability Discovery Using Machine Learning Techniques,” in Artificial Intelligence and Security, Cham, 2019, pp. 308–317. doi: 10/gg58hr.

M. Jimenez, R. Rwemalika, M. Papadakis, F. Sarro, Y. Le Traon, and M. Harman, “The importance of accounting for real-world labelling when predicting software vulnerabilities,” in Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, New York, NY, USA, Aug. 2019, pp. 695–705. doi: 10/gk52p5.

J. Kronjee, A. Hommersom, and H. Vranken, “Discovering software vulnerabilities using data-flow analysis and machine learning,” in Proceedings of the 13th International Conference on Availability, Reliability and Security, New York, NY, USA, Aug. 2018, pp. 1–10. doi: 10/gk52pv.

G. Lin, W. Xiao, J. Zhang, and Y. Xiang, “Deep Learning-Based Vulnerable Function Detection: a benchmark,” in Information and Communications Security, Cham, 2020, pp. 219–232. doi: 10/gk52p2.

Z. Li and Y. Shao, “A Survey of Feature Selection for Vulnerability Prediction Using Feature-based Machine Learning,” in Proceedings of the 2019 11th International Conference on Machine Learning and Computing, New York, NY, USA, Feb. 2019, pp. 36–42. doi: 10/gk52p6.

Z. Li et al., “VulDeePecker: a deep learning-based system for vulnerability detection,” Proceedings 2018 Network and Distributed System Security Symposium, 2018, doi: 10/gf96vp.

H. N. Ngoc, H. N. Viet, and T. Uehara, “An Extended Benchmark System of Word Embedding Methods for Vulnerability Detection,” in the 4th International Conference on Future Networks and Distributed Systems (ICFNDS), New York, NY, USA, Nov. 2020, pp. 1–8. doi: 10/gk52pw.

S. Omri and C. Sinz, “Deep Learning for Software Defect Prediction: a survey,” in Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops, New York, NY, USA, Jun. 2020, pp. 209–214. doi: 10/gk52p7.

A. Pechenkin and R. Demidov, “Applying Deep Learning and Vector Representation for Software Vulnerabilities Detection,” in Proceedings of the 11th International Conference on Security of Information and Networks, New York, NY, USA, Sep. 2018, pp. 1–6. doi: 10/gk52px.

M. Pradel and K. Sen, “DeepBugs: a learning approach to name-based bug detection,” Proc. ACM Program. Lang., vol. 2, no. OOPSLA, p. 147:1-147:25, Oct. 2018, doi: 10/ggwxh2.

W. Wang, G. Li, S. Shen, X. Xia, and Z. Jin, “Modular Tree Network for Source Code Representation Learning,” ACM Trans. Softw. Eng. Methodol., vol. 29, no. 4, p. 31:1-31:23, Sep. 2020, doi: 10/ghncgc.

W. Xiaomeng, Z. Tao, W. Runpu, X. Wei, and H. Changyu, “CPGVA: code property graph based vulnerability analysis by deep learning,” in 2018 10th International Conference on Advanced Infocomm Technology (ICAIT), Aug. 2018, pp. 184–188. doi: 10/gk52p4.

F. Yamaguchi, A. Maier, H. Gascon, and K. Rieck, “Automatic Inference of Search Patterns for Taint-Style Vulnerabilities,” in 2015 IEEE Symposium on Security and Privacy, May 2015, pp. 797–812. doi: 10/gfphc2.

H. Yan, Y. Sui, S. Chen, and J. Xue, “Machine-Learning-Guided Typestate Analysis for Static Use-After-Free Detection,” in Proceedings of the 33rd Annual Computer Security Applications Conference, New York, NY, USA, Dec. 2017, pp. 42–54. doi: 10/gf9n4s.

15.2.2 Round I

X. Cheng et al., “Static Detection of Control-Flow-Related Vulnerabilities Using Graph Embedding,” in 2019 24th International Conference on Engineering of Complex Computer Systems (ICECCS), Nov. 2019, pp. 41–50. doi: 10/gk52qc.

M.-J. Choi, S. Jeong, H. Oh, and J. Choo, “End-to-end prediction of buffer overruns from raw source code via neural memory networks,” in Proceedings of the 26th International Joint Conference on Artificial Intelligence, Melbourne, Australia, Aug. 2017, pp. 1546–1553.

Y. Fang, S. Han, C. Huang, and R. Wu, “TAP: a static analysis model for php vulnerabilities based on token and deep learning technology,” PLOS ONE, vol. 14, no. 11, p. e0225196, Nov. 2019, doi: 10/gk52p8.

G. Huang, Y. Li, Q. Wang, J. Ren, Y. Cheng, and X. Zhao, “Automatic Classification Method for Software Vulnerability Based on Deep Neural Network,” IEEE Access, vol. 7, pp. 28291–28298, 2019, doi: 10/gk52qd.

Z. Jin and Y. Yu, “Current and Future Research of Machine Learning Based Vulnerability Detection,” in 2018 Eighth International Conference on Instrumentation Measurement, Computer, Communication and Control (IMCCC), Jul. 2018, pp. 1562–1566. doi: 10/ghgn9t.

X. Li, L. Wang, Y. Xin, Y. Yang, and Y. Chen, “Automated Vulnerability Detection in Source Code Using Minimum Intermediate Representation Learning,” Applied Sciences, vol. 10, no. 5, p. 1692, Jan. 2020, doi: 10/gk52p9.

Z. Li, D. Zou, J. Tang, Z. Zhang, M. Sun, and H. Jin, “A Comparative Study of Deep Learning-Based Vulnerability Detection System,” IEEE Access, vol. 7, pp. 103184–103197, 2019, doi: 10/ggsskk.

G. Lin, S. Wen, Q.-L. Han, J. Zhang, and Y. Xiang, “Software Vulnerability Detection Using Deep Neural Networks: a survey,” Proceedings of the IEEE, vol. 108, no. 10, pp. 1825–1848, Oct. 2020, doi: 10/gg58hp.

S. Liu et al., “CD-VulD: cross-domain vulnerability discovery based on deep domain adaptation,” IEEE Transactions on Dependable and Secure Computing, pp. 1–1, 2020, doi: 10/gk52qb.

R. Rabheru, H. Hanif, and S. Maffeis, “DeepTective: detection of php vulnerabilities using hybrid graph neural networks,” Proceedings of the 36th Annual ACM Symposium on Applied Computing, pp. 1687–1690, Mar. 2021, Accessed: Jun. 02, 2021. [Online]. Available: https://doi.org/10.1145/3412841.3442132

N. Saccente, J. Dehlinger, L. Deng, S. Chakraborty, and Y. Xiong, “Project Achilles: a prototype tool for static method-level vulnerability detection of java source code using a recurrent neural network,” in 2019 34th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW), Nov. 2019, pp. 114–121. doi: 10/ggssjc.

Y. Wang, K. Wang, F. Gao, and L. Wang, “Learning semantic program embeddings with graph interval neural network,” Proc. ACM Program. Lang., vol. 4, no. OOPSLA, p. 137:1-137:27, Nov. 2020, doi: 10/ghwt5j.

F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, “Modeling and Discovering Vulnerabilities with Code Property Graphs,” in 2014 IEEE Symposium on Security and Privacy, May 2014, pp. 590–604. doi: 10/gf5b7d.

Y. Zhao, X. Du, P. Krishnan, and C. Cifuentes, “Buffer overflow detection for C programs is hard to learn,” in Companion Proceedings for the ISSTA/ECOOP 2018 Workshops, New York, NY, USA, Jul. 2018, pp. 8–9. doi: 10/gk52qf.

D. Zou, S. Wang, S. Xu, Z. Li, and H. Jin, “ μ VulDeePecker: a deep learning-based system for multiclass vulnerability detection,” IEEE Transactions on Dependable and Secure Computing, pp. 1–1, 2019, doi: 10/gkgd74.

15.2.3 Round II

M. Ahmadi, R. M. Farkhani, R. Williams, and L. Lu, “Finding Bugs Using Your Own Code: detecting functionally-similar yet inconsistent code,” presented at the 30th {USENIX} Security Symposium ({USENIX} Security 21), 2021. Accessed: Jun. 18, 2021. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/ahmadi

M. Alenezi, M. Zagane, and Y. Javed, “EFFICIENT DEEP FEATURES LEARNING FOR VULNERABILITY DETECTION USING CHARACTER N-GRAM EMBEDDING,” JJCIT, no. 0, p. 1, 2020, doi: 10/gk52qz.

H. Alves, B. Fonseca, and N. Antunes, “Experimenting Machine Learning Techniques to Predict Vulnerabilities,” in 2016 Seventh Latin-American Symposium on Dependable Computing (LADC), Oct. 2016, pp. 151–156. doi: 10/gkgf5k.

W. An, L. Chen, J. Wang, G. Du, G. Shi, and D. Meng, “AVDHRAM: automated vulnerability detection based on hierarchical representation and attention mechanism,” in 2020 IEEE Intl Conf on Parallel Distributed Processing with Applications, Big Data Cloud Computing, Sustainable Computing Communications, Social Computing Networking (ISPA/BDCloud/SocialCom/SustainCom), Dec. 2020, pp. 337–344. doi: 10/gk52qt.

T. Chappelly, C. Cifuentes, P. Krishnan, and S. Gevay, “Machine learning for finding bugs: an initial report,” in 2017 IEEE Workshop on Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE), Klagenfurt, Austria, Feb. 2017, pp. 21–26. doi: 10/gg68bq.

G. Fan, X. Diao, H. Yu, K. Yang, and L. Chen, “Software Defect Prediction via Attention-Based Recurrent Neural Network,” Scientific Programming, vol. 2019, p. e6230953, Apr. 2019, doi: 10/gk52qx.

A. Fidalgo, I. Medeiros, P. Antunes, and N. Neves, “Towards a Deep Learning Model for Vulnerability Detection on Web Application Variants,” in 2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), Oct. 2020, pp. 465–476. doi: 10/gk52q3.

A. Figueiredo, T. Lide, D. Matos, and M. Correia, “MERLIN: multi-language web vulnerability detection,” in 2020 IEEE 19th International Symposium on Network Computing and Applications (NCA), Nov. 2020, pp. 1–9. doi: 10/gk52q2.

N. Guo, X. Li, H. Yin, and Y. Gao, “VulHunter: an automated vulnerability detection system based on deep learning and bytecode,” in Information and Communications Security, Cham, 2020, pp. 199–218. doi: 10/gk52qw.

G. Jie, K. Xiao-Hui, and L. Qiang, “Survey on Software Vulnerability Analysis Method Based on Machine Learning,” in 2016 IEEE First International Conference on Data Science in Cyberspace (DSC), Jun. 2016, pp. 642–647. doi: 10/gk52q4.

S. Kim, S. Woo, H. Lee, and H. Oh, “VUDDY: a scalable approach for vulnerable code clone discovery,” in 2017 IEEE Symposium on Security and Privacy (SP), May 2017, pp. 595–614. doi: 10/ggssk5.

G. Lin et al., “Cross-Project Transfer Representation Learning for Vulnerable Function Discovery,” IEEE Transactions on Industrial Informatics, vol. 14, no. 7, pp. 3289–3297, Jul. 2018, doi: 10/gdwfhd.

G. Lin, J. Zhang, W. Luo, L. Pan, and Y. Xiang, “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, Oct. 2017, pp. 2539–2541. doi: 10/gkgf5h.

Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, and Z. Chen, “SySeVR: a framework for using deep learning to detect software vulnerabilities,” IEEE Transactions on Dependable and Secure Computing, pp. 1–1, 2021, doi: 10/gjwbqv.

S. Chakraborty, R. Krishna, Y. Ding, and B. Ray, “Deep Learning based Vulnerability Detection: are we there yet,” IEEE Transactions on Software Engineering, pp. 1–1, 2021, doi: 10/gk52qr.

C. B. Şahin, Ö. B. Dinler, and L. Abualigah, "Prediction of software vulnerability based deep symbiotic genetic algorithms: phenotyping of dominant-features," Appl Intell, Mar. 2021, doi: 10/gjmfrr.

S. Wang, T. Liu, and L. Tan, "Automatically learning semantic features for defect prediction," in Proceedings of the 38th International Conference on Software Engineering, New York, NY, USA, May 2016, pp. 297–308. doi: 10/gk36rj.

L. Wang, X. Li, R. Wang, Y. Xin, M. Gao, and Y. Chen, "PreNNsem: a heterogeneous ensemble learning framework for vulnerability detection in software," Applied Sciences, vol. 10, no. 22, p. 7954, Jan. 2020, doi: 10/gk52qv.

F. Wu, J. Wang, J. Liu, and W. Wang, "Vulnerability detection with deep learning," in 2017 3rd IEEE International Conference on Computer and Communications (ICCC), Dec. 2017, pp. 1298–1302. doi: 10/gk52q5.

P. Zeng, G. Lin, L. Pan, Y. Tai, and J. Zhang, "Software Vulnerability Analysis and Discovery Using Deep Learning Techniques: a survey," IEEE Access, vol. 8, pp. 197158–197172, 2020, doi: 10/gk52qs.

W. Zheng, J. Gao, X. Wu, Y. Xun, G. Liu, and X. Chen, "An Empirical Study of High-Impact Factors for Machine Learning-Based Vulnerability Detection," in 2020 IEEE 2nd International Workshop on Intelligent Bug Fixing (IBF), Feb. 2020, pp. 26–34. doi: 10/ghrm2j.

15.2.4 Round III

L. Cui, Z. Hao, Y. Jiao, H. Fei, and X. Yun, "VulDetector: detecting vulnerabilities using weighted feature graph comparison," IEEE Transactions on Information Forensics and Security, vol. 16, pp. 2004–2017, 2021, doi: 10/gk52qm.

X. Duan et al., "VulSniper: focus your attention to shoot fine-grained vulnerabilities," in Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19, 2019, pp. 4665–4671. doi: 10/gk52qq.

H. Feng, X. Fu, H. Sun, H. Wang, and Y. Zhang, "Efficient Vulnerability Detection based on abstract syntax tree and Deep Learning," in IEEE INFOCOM 2020 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS), Jul. 2020, pp. 722–727. doi: 10/gk52qk.

R. Li, C. Feng, X. Zhang, and C. Tang, "A Lightweight Assisted Vulnerability Discovery Method Using Deep Neural Networks," IEEE Access, vol. 7, pp. 80079–80092, 2019, doi: 10/ggssht.

G. Lin et al., "Software Vulnerability Discovery via Learning Multi-domain Knowledge Bases," IEEE Transactions on Dependable and Secure Computing, pp. 1–1, 2019, doi: 10/ghk5jn.

S. Liu, G. Lin, Q.-L. Han, S. Wen, J. Zhang, and Y. Xiang, "DeepBalance: deep-learning and fuzzy oversampling for vulnerability detection," IEEE Transactions on Fuzzy Systems, vol. 28, no. 7, pp. 1329–1343, Jul. 2020, doi: 10/gk52qh.

Z. Li, D. Zou, S. Xu, Z. Chen, Y. Zhu, and H. Jin, "VulDeeLocator: a deep learning-based fine-grained vulnerability detector," IEEE Transactions on Dependable and Secure Computing, pp. 1–1, 2021, doi: 10/gjwbq7.

S. Ma, F. Thung, D. Lo, C. Sun, and R. H. Deng, "VuRLE: automatic vulnerability detection and repair by learning from examples," in Computer Security – ESORICS 2017, Cham, 2017, pp. 229–246. doi: 10/ggzsh8.

I. Medeiros, N. Neves, and M. Correia, "DEKANT: a static analysis tool that learns to detect web application vulnerabilities," in Proceedings of the 25th International Symposium on Software Testing and Analysis, New York, NY, USA, Jul. 2016, pp. 1–11. doi: 10/gk52qp.

V. Nguyen et al., "Deep Domain Adaptation for Vulnerable Code Function Identification," in 2019 International Joint Conference on Neural Networks (IJCNN), Jul. 2019, pp. 1–8. doi: 10/gk52qn.

H. Zhang, Y. Bi, H. Guo, W. Sun, and J. Li, “ISVSF: intelligent vulnerability detection against java via sentence-level pattern exploring,” IEEE Systems Journal, pp. 1–12, 2021, doi: 10/gk52qj.

W. Zheng, A. O. Abdallah Semasaba, X. Wu, S. A. Agyemang, T. Liu, and Y. Ge, “Representation vs. Model: what matters most for source code vulnerability detection,” in 2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), Mar. 2021, pp. 647–653. doi: 10/gk52qg.

Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, “Devign: effective vulnerability identification by learning comprehensive program semantics via graph neural networks,” arXiv:1909.03496 [cs, stat], Sep. 2019, Accessed: Jun. 24, 2021. [Online]. Available: http://arxiv.org/abs/1909.03496

15.2.5 Round IV

A. Xu, T. Dai, H. Chen, Z. Ming, and W. Li, “Vulnerability Detection for Source Code Using Contextual LSTM,” in 2018 5th International Conference on Systems and Informatics (ICSAI), Nov. 2018, pp. 1225–1230. doi: 10.1109/ICSAI.2018.8599360.

X. Ban, S. Liu, C. Chen, and C. Chua, “A performance evaluation of deep-learnt features for software vulnerability detection,” Concurrency and Computation: Practice and Experience, vol. 31, no. 19, p. e5103, 2019, doi: 10.1002/cpe.5103.

R. Russell et al., “Automated Vulnerability Detection in Source Code Using Deep Representation Learning,” in 2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA), Dec. 2018, pp. 757–762. doi: 10.1109/ICMLA.2018.00120.

Z. Bilgin, M. A. Ersoy, E. U. Soykan, E. Tomur, P. Çomak, and L. Karaçay, “Vulnerability Prediction from Source Code Using Machine Learning,” IEEE Access, vol. 8, pp. 150672–150684, 2020, doi: 10.1109/ACCESS.2020.3016774.

15.3 Guideline for the interviews with the experts

Listed are the questions which were defined for the expert interviews. The questions were in German as the interview partners were German. A translation of the question to English is also provided.

Question No.	*Question in German*	*Question in English*
1	Welche Erfahrungen haben Sie mit den Themen C und Security von C-Code?	What experience do you have with C and security of C code?
2	In welchem Umfeld setzen Sie die Programmiersprache C ein?	In which context do you use the programming language C?
3	Welche C-Versionen setzen Sie in Ihren Projekten ein? Auswahlmöglichkeiten: C99, C11, C17, andere	Which C versions do you use in your projects? Selection options: C99, C11, C17, other
4	C-Versionen in anderen Entwicklerteams? Auswahlmöglichkeiten C99, C11, C17, andere	Which C version do other software development teams in your organization use? Selection options: C99, C11, C17, other
5	Nutzen Sie statische Programmanalyse-Werkzeuge (SAST Tools)?	Do you use static software analysis tools (SAST tools)?
6	Wenn ja, welche? Für welche Sprachen?	If so, which tools? For what programming languages?
7	Welche Tools sind besonders gut für C geeignet?	Which tools are especially well equipped for the C programming language?
8	Welche Tools sind besonders gut für C++ geeignet?	Which tools are especially well equipped for the C++ programming language?
9	Welche Vorteile bieten SAST Tools?	What advantages do SAST tools offer?

Question No.	***Question in German***	***Question in English***
10	Welche Probleme treten dabei auf?	What problems do these tools suffer from?
11	Beim Einsatz SAST-Tools: Schreiben Sie eigene Regeln hierfür?	If SAST tools are employed: Do you define custom rules to this end?
12	Welche Formate bzw. Eingabemöglichkeiten von Regeln kennen Sie (z.B. XML, User Interface)?	What formats, respectively what options regarding the input of rules do you know of (e.g., XML, user interface)?
13	Welche Fälle werden meist gut detektiert?	What types of errors are usually well detected?
14	Kennen Sie bzw. können Sie sich konkrete Fälle (im C-Code) vorstellen, die nicht durch statische Werkzeuge detektiert werden bzw. im Gesamtrauschen untergehen (sog. Fehler-Use Cases)? Bitte nennen Sie konkrete Beispiele! (Dies kann auch im Nachgang mit Kolleginnen und Kollegen passieren.)	Can you think of any concrete examples (of C-code) that cannot be detected using SAST tools and are overshadowed by background noise (so-called error use cases), respectively? If so, please state these examples! (This may also be done after the interview with the help of colleagues.)
15	Kennen Sie ggf. Quellen (z.B. Testrepositories, Publikationen), in denen solche Fehler-Use-Cases dokumentiert sind?	Do you know of any sources (e.g., repositories of test data, publications) that document such error use cases?
16	In welchem Format liegen solche Fehler-Uses-Cases?	In what format are such error use cases present?
17	Welche Anforderungen haben Sie an ein Tool, das z.B. Regeln mittels ML lernen kann?	What are your requirements concerning a tool that, for instance, infers rules using machine learning techniques?
18	Kennen Sie bereits (kommerzielle) statische Analysetools oder wissenschaftliche Prototypen, die eine solche Funktionalität aufweisen?	Do you know of any (commercial) static analysis tools or scientific prototypes that exhibit such functionality?

Bibliography

[1] J. Kari, “Rice’s theorem for the limit sets of cellular automata,” *Theoretical Computer Science*, vol. 127, no. 2, pp. 229–254, May 1994, doi: 10.1016/0304-3975(94)90041-8.

[2] B. Chess and J. West, *Secure programming with static analysis*, 1st ed. Addison-Wesley Professional, 2007.

[3] G. McGraw, *Software security: Building security in*. Addison-Wesley Professional, 2006.

[4] Bundesamt für Sicherheit in der Informationstechnik, “Künstliche Intelligenz,” *Bundesamt für Sicherheit in der Informationstechnik*, Nov. 15, 2021. https://www.bsi.bund.de/DE/Themen/Unternehmen-und-Organisationen/Informationen-und-Empfehlungen/Kuenstliche-Intelligenz/KI.html;jsessionid=F743D5614214A08BDB0E5C921F5949BB.internet482?nn=129146 (accessed Nov. 15, 2021).

[5] AI HLEG, “A definition of AI: Main capabilities and scientific disciplines,” Apr. 2019.

[6] NATO, “NATO Review - An Artificial Intelligence Strategy for NATO,” *NATO Review*, Oct. 25, 2021. https://www.nato.int/docu/review/articles/2021/10/25/an-artificial-intelligence-strategy-for-nato/index.html (accessed Nov. 15, 2021).

[7] N. A. and R. A. Office of the Federal Register, “DCPD-202000870 - Executive Order 13960-Promoting the Use of Trustworthy Artificial Intelligence in the Federal Government,” *govinfo.gov*, Dec. 03, 2020. https://www.govinfo.gov/app/details/https%3A%2F%2Fwww.govinfo.gov%2Fapp%2Fdetails%2FDCPD-202000870 (accessed Nov. 15, 2021).

[8] European Comission, “Ethics guidelines for trustworthy AI | Shaping Europe’s digital future,” *DG Connect*, Apr. 08, 2019. https://digital-strategy.ec.europa.eu/en/library/ethics-guidelines-trustworthy-ai (accessed Nov. 15, 2021).

[9] A. M. Turing, “I.—COMPUTING MACHINERY AND INTELLIGENCE,” *Mind*, vol. LIX, no. 236, pp. 433–460, Oct. 1950, doi: 10/b262dj.

[10] A. Burkov, *The hundred-page machine learning book*. 2019.

[11] R. Bellman, “On the Theory of Dynamic Programming,” *Proceedings of the National Academy of Sciences*, vol. 38, no. 8, pp. 716–719, Aug. 1952, doi: 10/b9p5n7.

[12] J. Fürnkranz, T. Scheffer, and M. Spiliopoulou, *Machine Learning: ECML 2006: 17th European Conference on Machine Learning, Berlin, Germany, September 18-22, 2006, Proceedings*. Springer, 2006.

[13] A. C. Tsoi, “Recurrent neural network architectures: An overview,” in *Adaptive Processing of Sequences and Data Structures: International Summer School on Neural Networks “E.R. Caianiello” Vietri sul Mare, Salerno, Italy September 6–13, 1997 Tutorial Lectures*, C. L. Giles and M. Gori, Eds. Berlin, Heidelberg: Springer, 1998, pp. 1–26. doi: 10.1007/BFb0053993.

[14] G. Chen, “A Gentle Tutorial of Recurrent Neural Network with Error Backpropagation,” *arXiv:1610.02583 [cs]*, Jan. 2018, Accessed: Oct. 27, 2021. [Online]. Available: http://arxiv.org/abs/1610.02583

[15] R. Pascanu, T. Mikolov, and Y. Bengio, “On the difficulty of training recurrent neural networks,” in *Proceedings of the 30th International Conference on Machine Learning*, May 2013, pp. 1310–1318. Accessed: Oct. 01, 2021. [Online]. Available: https://proceedings.mlr.press/v28/pascanu13.html

[16] O. Calin, *Deep learning architectures: A mathematical approach*. Springer International Publishing, 2020. [Online]. Available: https://link.springer.com/book/10.1007/978-3-030-36721-3

[17] S. Hochreiter and J. Schmidhuber, “Long Short-Term Memory,” *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, Nov. 1997, doi: 10/bxd65w.

[18] S. Varsamopoulos, K. Bertels, and C. Almudever, *Designing neural network based decoders for surface codes*. 2018.

[19] K. Cho *et al.*, “Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation,” *arXiv:1406.1078 [cs, stat]*, Sep. 2014, Accessed: Nov. 15, 2021. [Online]. Available: http://arxiv.org/abs/1406.1078

[20] M. Schuster and K. K. Paliwal, “Bidirectional recurrent neural networks,” *IEEE Transactions on Signal Processing*, vol. 45, no. 11, pp. 2673–2681, Nov. 1997, doi: 10/cfqwr6.

[21] G. Lin, W. Xiao, J. Zhang, and Y. Xiang, “Deep Learning-Based Vulnerable Function Detection: a benchmark,” in *Information and Communications Security*, Cham, 2020, pp. 219–232. doi: 10/gk52p2.

[22] W. Zheng, J. Gao, X. Wu, Y. Xun, G. Liu, and X. Chen, “An Empirical Study of High-Impact Factors for Machine Learning-Based Vulnerability Detection,” in *2020 IEEE 2nd International Workshop on Intelligent Bug Fixing (IBF)*, Feb. 2020, pp. 26–34. doi: 10/ghrm2j.

[23] Z. Li, D. Zou, J. Tang, Z. Zhang, M. Sun, and H. Jin, “A Comparative Study of Deep Learning-Based Vulnerability Detection System,” *IEEE Access*, vol. 7, pp. 103184–103197, 2019, doi: 10/ggsskk.

[24] W. L. Hamilton, “Graph Representation Learning,” *Synthesis Lectures on Artificial Intelligence and Machine Learning*, vol. 14, no. 3, pp. 1–159, Sep. 2020, doi: 10.2200/S01045ED1V01Y202009AIM046.

[25] P. Flach, *Machine learning: The art and science of algorithms that make sense of data*. Cambridge University Press, 2012. [Online]. Available: https://www.cambridge.org/de/academic/subjects/computer-science/pattern-recognition-and-machine-learning/machine-learning-art-and-science-algorithms-make-sense-data?format=HB&isbn=9781107096394

[26] T. M. Mitchell, *Machine Learning*. New York: McGraw-Hill, 1997.

[27] P. Langley and M. B. Morgan, *Elements of machine learning*. Morgan Kaufmann, 1996.

[28] S. Shalev-Shwartz and S. Ben-David, *Understanding machine learning: From theory to algorithms*. Cambridge University Press, 2014.

[29] C. M. Bishop, *Pattern recognition and machine learning*. Springer, 2006.

[30] R. . S. Sutton and A. G. Barto, *Reinforcement learning, second edition: An introduction*. MIT Press, 2018.

[31] J. Graf, “Information Flow Control with System Dependence Graphs - Improving Modularity, Scalability and Precision for Object Oriented Languages,” 2016, doi: 10.5445/IR/1000068211.

[32] A. W. Appel, *Modern Compiler Implementation in Java: Basic Techniques*. USA: Cambridge University Press, 1997.

[33] P. Lam, E. Bodden, O. Lhoták, and L. Hendren, “The Soot framework for Java program analysis: a retrospective,” in *Cetus users and compiler infastructure workshop (CETUS 2011)*, 2011, vol. 15.

[34] J. Dolby and M. Sridharan, “Static and Dynamic Program Analysis Using WALA,” *PLDI 2010 Tutorial (T.J. Watson Libraries for Analysis)*, Oct. 06, 2010. http://wala.sourceforge.net/files/PLDI_WALA_Tutorial.pdf (accessed Sep. 04, 2021).

[35] C. Lattner and V. Adve, “LLVM: A compilation framework for lifelong program analysis & transformation,” in *Proceedings of the international symposium on code generation and optimization: Feedback-directed and runtime optimization*, USA, 2004, p. 75.

[36] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, “Modeling and Discovering Vulnerabilities with Code Property Graphs,” in *2014 IEEE Symposium on Security and Privacy*, May 2014, pp. 590–604. doi: 10/gf5b7d.

[37] S. Horwitz, T. Reps, and D. Binkley, “Interprocedural slicing using dependence graphs,” *ACM Trans. Program. Lang. Syst.*, vol. 12, no. 1, pp. 26–60, Jan. 1990, doi: 10.1145/77606.77608.

[38] P. Anderson and T. Teitelbaum, “Software inspection using codesurfer,” in *In workshop on inspection in software engineering*, 2001.

[39] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, techniques, and tools (2nd edition)*. USA: Addison-Wesley Longman Publishing Co., Inc., 2006.

[40] J. Krinke, “Advanced Slicing of Sequential and Concurrent Programs,” PhD Thesis, Universität Passau, 2003. [Online]. Available: http://www.dcs.kcl.ac.uk/staff/krinke/publications.php

[41] S. Horwitz, T. Reps, and D. Binkley, “Interprocedural slicing using dependence graphs,” *SIGPLAN Not.*, vol. 23, no. 7, pp. 35–46, Jun. 1988, doi: 10.1145/960116.53994.

[42] F. Yamaguchi, A. Maier, H. Gascon, and K. Rieck, “Automatic Inference of Search Patterns for Taint-Style Vulnerabilities,” in *2015 IEEE Symposium on Security and Privacy*, May 2015, pp. 797–812. doi: 10/gfphc2.

[43] D. Arp *et al.*, “Dos and Don’ts of Machine Learning in Computer Security,” *arXiv:2010.09470 [cs]*, Oct. 2020, Accessed: Sep. 02, 2021. [Online]. Available: http://arxiv.org/abs/2010.09470

[44] S. Wang, T. Liu, and L. Tan, “Automatically learning semantic features for defect prediction,” in *Proceedings of the 38th International Conference on Software Engineering*, New York, NY, USA, May 2016, pp. 297–308. doi: 10/gk36rj.

[45] Z. Li *et al.*, “VulDeePecker: a deep learning-based system for vulnerability detection,” *Proceedings 2018 Network and Distributed System Security Symposium*, 2018, doi: 10/gf96vp.

[46] R. Russell *et al.*, “Automated Vulnerability Detection in Source Code Using Deep Representation Learning,” in *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*, Dec. 2018, pp. 757–762. doi: 10.1109/ICMLA.2018.00120.

[47] W. An, L. Chen, J. Wang, G. Du, G. Shi, and D. Meng, “AVDHRAM: automated vulnerability detection based on hierarchical representation and attention mechanism,” in *2020 IEEE Intl Conf on Parallel Distributed Processing with Applications, Big Data Cloud Computing, Sustainable Computing Communications, Social Computing Networking (ISPA/BDCloud/SocialCom/SustainCom)*, Dec. 2020, pp. 337–344. doi: 10/gk52qt.

[48] W. Xiaomeng, Z. Tao, W. Runpu, X. Wei, and H. Changyu, “CPGVA: code property graph based vulnerability analysis by deep learning,” in *2018 10th International Conference on Advanced Infocomm Technology (ICAIT)*, Aug. 2018, pp. 184–188. doi: 10/gk52p4.

[49] V. Nguyen *et al.*, “Deep Domain Adaptation for Vulnerable Code Function Identification,” in *2019 International Joint Conference on Neural Networks (IJCNN)*, Jul. 2019, pp. 1–8. doi: 10/gk52qn.

[50] H. Feng, X. Fu, H. Sun, H. Wang, and Y. Zhang, “Efficient Vulnerability Detection based on abstract syntax tree and Deep Learning,” in *IEEE INFOCOM 2020 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, Jul. 2020, pp. 722–727. doi: 10/gk52qk.

[51] X. Duan *et al.*, “VulSniper: focus your attention to shoot fine-grained vulnerabilities,” in *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19*, 2019, pp. 4665–4671. doi: 10/gk52qq.

[52] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, and Z. Chen, “SySeVR: a framework for using deep learning to detect software vulnerabilities,” *IEEE Transactions on Dependable and Secure Computing*, pp. 1–1, 2021, doi: 10/gjwbqv.

[53] D. Zou, S. Wang, S. Xu, Z. Li, and H. Jin, “uVulDeePecker: a deep learning-based system for multiclass vulnerability detection,” *IEEE Transactions on Dependable and Secure Computing*, pp. 1–1, 2019, doi: 10/gkgd74.

[54] S. Chakraborty, R. Krishna, Y. Ding, and B. Ray, “Deep Learning based Vulnerability Detection: are we there yet,” *IEEE Transactions on Software Engineering*, pp. 1–1, 2021, doi: 10/gk52qr.

[55] X. Cheng, H. Wang, J. Hua, G. Xu, and Y. Sui, “DeepWukong: statically detecting software vulnerabilities using deep graph neural network,” *ACM Trans. Softw. Eng. Methodol.*, vol. 30, no. 3, p. 38:1-38:33, Apr. 2021, doi: 10/gk52pz.

[56] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, “Devign: effective vulnerability identification by learning comprehensive program semantics via graph neural networks,” *arXiv:1909.03496 [cs, stat]*, Sep. 2019, Accessed: Jun. 24, 2021. [Online]. Available: http://arxiv.org/abs/1909.03496

[57] A. Warnecke, D. Arp, C. Wressnegger, and K. Rieck, “Evaluating Explanation Methods for Deep Learning in Security,” *arXiv:1906.02108 [cs, stat]*, Apr. 2020, Accessed: Aug. 26, 2021. [Online]. Available: http://arxiv.org/abs/1906.02108

[58] J. Gläser and G. Laudel, *Experteninterviews und qualitative Inhaltsanalyse als Instrumente rekonstruierender Untersuchungen*, 4. Aufl. Wiesbaden: VS Verl. für Sozialwiss, 2010.

[59] A. Behrens, K. Sohr, R. Koschke, M. Schröer, and B. J. Berger, “A Survey on the Awareness, Understanding, and Attitudes Toward Security Patterns in Software Development,” *International Journal of Information Security (IJIS)*, 2021.

[60] S. Kim, S. Woo, H. Lee, and H. Oh, “VUDDY: A Scalable Approach for Vulnerable Code Clone Discovery,” in *2017 IEEE Symposium on Security and Privacy (SP)*, May 2017, pp. 595–614. doi: 10.1109/SP.2017.62.

[61] M. Green, “Attack of the week: OpenSSL Heartbleed,” *A Few Thoughts on Cryptographic Engineering*, Apr. 08, 2014. https://blog.cryptographyengineering.com/2014/04/08/attack-of-the-week-openssl-heartbleed/ (accessed Oct. 11, 2021).

[62] Y. Xiao *et al.*, “MVP: Detecting vulnerabilities using patch-enhanced vulnerability signatures,” in *29th {USENIX} Security Symposium ({USENIX} Security 20)*, 2020, pp. 1165–1182.

[63] team teso and scut, “Exploiting Format String Vulnerabilities,” Sep. 2001.

[64] F. Yamaguchi, C. Wressnegger, H. Gascon, and K. Rieck, “Chucky: exposing missing checks in source code for vulnerability discovery,” in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, New York, NY, USA, Nov. 2013, pp. 499–510. doi: 10.1145/2508859.2516665.

[65] M. Ahmadi, R. M. Farkhani, R. Williams, and L. Lu, “Finding Bugs Using Your Own Code: detecting functionally-similar yet inconsistent code,” presented at the 30th {USENIX} Security Symposium ({USENIX} Security 21), 2021. Accessed: Jun. 18, 2021. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/ahmadi

[66] C. Paar and J. Pelzl, *Understanding Cryptography: A Textbook for Students and Practitioners*, 1st ed. Springer Publishing Company, Incorporated, 2009.

[67] The MITRE Corporation, “CWE - CWE-416: Use After Free (4.5),” Jul. 19, 2006. https://cwe.mitre.org/data/definitions/416.html (accessed Sep. 04, 2021).

[68] K. Zhu, Y. Lu, and H. Huang, “Scalable Static Detection of Use-After-Free Vulnerabilities in Binary Code,” *IEEE Access*, vol. 8, pp. 78713–78725, 2020, doi: 10.1109/ACCESS.2020.2990197.

[69] The MITRE Corporation, “CWE - CWE-366: Race Condition within a Thread (4.5),” Jul. 19, 2006. https://cwe.mitre.org/data/definitions/366.html (accessed Oct. 11, 2021).

[70] Carnegie Mellon University, “SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition),” Carnegie Mellon University, SEI Administrative Agent

AFLCMC/PZM 20 Schilling Circle, Bldg. 1305, 3rd floor Hanscom AFB, MA 01731-2125, DM-0003560, 2016. [Online]. Available: SEI CERT C Coding Standard

[71] M. Turek, “Explainable Artificial Intelligence,” DARPA-BAA-16-53, Aug. 2016. Accessed: Nov. 17, 2021. [Online]. Available: https://www.darpa.mil/program/explainable-artificial-intelligence

[72] A.-M. Leventi-Peetz, T. Östreich, W. Lennartz, and K. Weber, “Scope and Sense of Explainability for AI-Systems,” *arXiv:2112.10551 [cs]*, vol. 294, pp. 291–308, 2022, doi: 10/gntt5s.

[73] T. Miller, “Explanation in Artificial Intelligence: Insights from the Social Sciences,” *arXiv:1706.07269 [cs]*, Aug. 2018, Accessed: Nov. 18, 2021. [Online]. Available: http://arxiv.org/abs/1706.07269

[74] H. Yuan, H. Yu, S. Gui, and S. Ji, “Explainability in Graph Neural Networks: A Taxonomic Survey,” *arXiv:2012.15445 [cs]*, Mar. 2021, Accessed: Aug. 26, 2021. [Online]. Available: http://arxiv.org/abs/2012.15445

[75] C. Molnar, *Interpretable machine learning*. Lulu. com, 2020.

[76] I. J. Goodfellow, J. Shlens, and C. Szegedy, “Explaining and Harnessing Adversarial Examples,” *arXiv:1412.6572 [cs, stat]*, Mar. 2015, Accessed: Nov. 18, 2021. [Online]. Available: http://arxiv.org/abs/1412.6572

[77] N. Antunes and M. Vieira, “On the Metrics for Benchmarking Vulnerability Detection Tools,” in *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, Jun. 2015, pp. 505–516. doi: 10.1109/DSN.2015.30.

[78] M. Jimenez, R. Rwemalika, M. Papadakis, F. Sarro, Y. Le Traon, and M. Harman, “The importance of accounting for real-world labelling when predicting software vulnerabilities,” in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, New York, NY, USA, Aug. 2019, pp. 695–705. doi: 10/gk52p5.

[79] H. M and S. M.N, “A Review on Evaluation Metrics for Data Classification Evaluations,” *IJDKP*, vol. 5, no. 2, pp. 01–11, Mar. 2015, doi: 10/ghn4np.

[80] F. S. Rizi, J. Schloetterer, and M. Granitzer, “Shortest Path Distance Approximation Using Deep Learning Techniques,” in *2018 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM)*, Aug. 2018, pp. 1007–1014. doi: 10/gjg4q4.

[81] VulDetProject and S. Chakraborty, “Deep Learning based Vulnerability Detection:Are We There Yet?” Oct. 10, 2021. Accessed: Oct. 12, 2021. [Online]. Available: https://github.com/VulDetProject/ReVeal

[82] F. Yamaguchi, D. Baker Effendi, and N. Schmidt, “Overview | Joern Documentation,” 2021. https://docs.joern.io/home (accessed Oct. 12, 2021).

[83] F. Yamaguchi, D. Baker Effendi, and N. Schmidt, “Joern - The Bug Hunter’s Workbench,” *Joern - The Bug Hunter’s Workbench*, 2021. https://joern.io/ (accessed Oct. 12, 2021).

[84] L. Kim and R. Russell, “Draper VDISC Dataset - Vulnerability Detection in Source Code,” Nov. 2018, Accessed: Oct. 12, 2021. [Online]. Available: https://osf.io/d45bw/

[85] D. Cournapeau, “scikit-learn: machine learning in Python — scikit-learn 1.0 documentation,” Jun. 2007. https://scikit-learn.org/stable/ (accessed Oct. 13, 2021).

[86] G. Brain Team, “TensorFlow,” *TensorFlow*, 2015. https://www.tensorflow.org/ (accessed Oct. 13, 2021).

[87] F. Chollet and Keras Team, “Keras: the Python deep learning API,” 2015. https://keras.io/ (accessed Oct. 13, 2021).

[88] LF Projects, LLC and MLflow Project, “MLflow - A platform for the machine learning lifecycle,” *MLflow*, Jun. 05, 2018. https://mlflow.org/ (accessed Oct. 13, 2021).

[89] D. Esteves *et al.*, “MEX vocabulary: a lightweight interchange format for machine learning experiments,” in *Proceedings of the 11th International Conference on Semantic Systems*, New York, NY, USA, Sep. 2015, pp. 169–176. doi: 10.1145/2814864.2814883.

[90] G. C. Publio *et al.*, “ML-Schema: Exposing the Semantics of Machine Learning with Schemas and Ontologies,” *arXiv:1807.05351 [cs, stat]*, Jul. 2018, Accessed: Oct. 20, 2021. [Online]. Available: http://arxiv.org/abs/1807.05351

[91] K. Petersen, R. Feldt, S. Mujtaba, and M. Mattsson, “Systematic mapping studies in software engineering,” in *Proceedings of the 12th international conference on Evaluation and Assessment in Software Engineering*, Swindon, GBR, Jun. 2008, pp. 68–77.

[92] C. Wohlin, “Guidelines for snowballing in systematic literature studies and a replication in software engineering,” in *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*, New York, NY, USA, May 2014, pp. 1–10. doi: 10.1145/2601248.2601268.

[93] C. Sammut and G. I. Webb, Eds., *Encyclopedia of machine learning*. New York ; London: Springer, 2010.

[94] The MITRE Corporation, “CWE - CWE List Version 4.5,” *Common Weakness Enumeration (CWE)*, Jul. 20, 2021. https://cwe.mitre.org/data/ (accessed Sep. 04, 2021).

[95] The MITRE Corporation, "CWE - CWE-664: Improper Control of a Resource Through its Lifetime (4.5)," Apr. 11, 2008. https://cwe.mitre.org/data/definitions/664.html (accessed Aug. 17, 2021).

[96] The MITRE Corporation, "CWE - CWE-707: Improper Neutralization (4.5)," Sep. 09, 2008. https://cwe.mitre.org/data/definitions/707.html (accessed Aug. 18, 2021).

[97] H. N. Ngoc, H. N. Viet, and T. Uehara, "An Extended Benchmark System of Word Embedding Methods for Vulnerability Detection," in *The 4th International Conference on Future Networks and Distributed Systems (ICFNDS)*, New York, NY, USA, Nov. 2020, pp. 1–8. doi: 10/gk52pw.

[98] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "SMOTE: Synthetic Minority Over-sampling Technique," *Journal of Artificial Intelligence Research*, vol. 16, pp. 321–357, Jun. 2002, doi: 10.1613/jair.953.

[99] B. Frenay and M. Verleysen, "Classification in the Presence of Label Noise: A Survey," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 25, no. 5, pp. 845–869, May 2014, doi: 10/f5zdgq.

[100] G. Patrini, A. Rozza, A. Menon, R. Nock, and L. Qu, "Making Deep Neural Networks Robust to Label Noise: a Loss Correction Approach," *arXiv:1609.03683 [cs, stat]*, Mar. 2017, Accessed: Aug. 18, 2021. [Online]. Available: http://arxiv.org/abs/1609.03683

[101] S. Liu, G. Lin, Q.-L. Han, S. Wen, J. Zhang, and Y. Xiang, "DeepBalance: deep-learning and fuzzy oversampling for vulnerability detection," *IEEE Transactions on Fuzzy Systems*, vol. 28, no. 7, pp. 1329–1343, Jul. 2020, doi: 10/gk52qh.

[102] M. Gori, G. Monfardini, and F. Scarselli, "A new model for learning in graph domains," in *Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005.*, Jul. 2005, vol. 2, pp. 729–734 vol. 2. doi: 10/cr2f33.

[103] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini, "The Graph Neural Network Model," *IEEE Transactions on Neural Networks*, vol. 20, no. 1, pp. 61–80, Jan. 2009, doi: 10/fgf4kh.

[104] J. Zhou *et al.*, "Graph neural networks: A review of methods and applications," *AI Open*, vol. 1, pp. 57–81, 2020, doi: 10/gjt96x.

[105] Y. Wang, K. Wang, F. Gao, and L. Wang, "Learning semantic program embeddings with graph interval neural network," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, p. 137:1-137:27, Nov. 2020, doi: 10/ghwt5j.

[106] J. Plenio, "Pferd Frisst Gras," Jul. 11, 2018. https://www.pexels.com/de-de/foto/pferd-frisst-gras-1574546/ (accessed Aug. 27, 2021).

[107] P. Dollar, "COCO - Common Objects in Context," 2017. https://cocodataset.org/#home (accessed Aug. 26, 2021).

[108] G. Montavon, "Tutorial: Implementing Layer-Wise Relevance Propagation," *gmontavon / LRP tutorial · GitLab*, Jul. 05, 2021. https://git.tu-berlin.de/gmontavon/lrp-tutorial (accessed Aug. 25, 2021).

[109] S. Bach, A. Binder, G. Montavon, F. Klauschen, K.-R. Müller, and W. Samek, "On Pixel-Wise Explanations for Non-Linear Classifier Decisions by Layer-Wise Relevance Propagation," *PLOS ONE*, vol. 10, no. 7, p. e0130140, Jul. 2015, doi: 10/gcgmcp.

[110] S. Cao, X. Sun, L. Bo, Y. Wei, and B. Li, "BGNN4VD: Constructing Bidirectional Graph Neural-Network for Vulnerability Detection," *Information and Software Technology*, vol. 136, p. 106576, Aug. 2021, doi: 10/gmbqfr.

[111] T. Ganz, M. Härterich, A. Warnecke, and K. Rieck, "Explaining Graph Neural Networks for Vulnerability Discovery," in *Proceedings of the 14th ACM Workshop on Artificial Intelligence and Security*, New York, NY, USA, Nov. 2021, pp. 145–156. doi: 10/gnc7rb.

[112] P. Zeng, G. Lin, L. Pan, Y. Tai, and J. Zhang, "Software Vulnerability Analysis and Discovery Using Deep Learning Techniques: a survey," *IEEE Access*, vol. 8, pp. 197158–197172, 2020, doi: 10/gk52qs.

[113] M. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to Represent Programs with Graphs," *arXiv:1711.00740 [cs]*, May 2018, Accessed: Aug. 19, 2021. [Online]. Available: http://arxiv.org/abs/1711.00740

[114] F. Wu, J. Wang, J. Liu, and W. Wang, "Vulnerability detection with deep learning," in *2017 3rd IEEE International Conference on Computer and Communications (ICCC)*, Dec. 2017, pp. 1298–1302. doi: 10/gk52q5.

[115] W. Zheng, A. O. Abdallah Semasaba, X. Wu, S. A. Agyemang, T. Liu, and Y. Ge, "Representation vs. Model: what matters most for source code vulnerability detection," in *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, Mar. 2021, pp. 647–653. doi: 10/gk52qg.

[116] F. Fischer *et al.*, "Stack Overflow Considered Helpful! Deep Learning Security Nudges Towards Stronger Cryptography," presented at the 28th {USENIX} Security Symposium ({USENIX} Security 19), 2019, pp. 339–356. Accessed: May 26, 2021. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/fischer

[117] K. Chen, S. Guo, T. Zhang, X. Xie, and Y. Liu, "Stealing Deep Reinforcement Learning Models for Fun and Profit," in *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*, New York, NY, USA, May 2021, pp. 307–319. doi: 10.1145/3433210.3453090.

[118] X. Ban, S. Liu, C. Chen, and C. Chua, "A performance evaluation of deep-learnt features for software vulnerability detection," *Concurrency and Computation: Practice and Experience*, vol. 31, no. 19, p. e5103, 2019, doi: 10.1002/cpe.5103.

[119] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Trans. Program. Lang. Syst.*, vol. 9, no. 3, pp. 319–349, Jul. 1987, doi: 10.1145/24039.24041.

[120] U. Schöning, "Graph isomorphism is in the low hierarchy," *Journal of Computer and System Sciences*, vol. 37, no. 3, pp. 312–323, Dec. 1988, doi: 10.1016/0022-0000(88)90010-4.

[121] A. Dutta and H. Sahbi, "Stochastic Graphlet Embedding," *IEEE Trans. Neural Netw. Learning Syst.*, vol. 30, no. 8, pp. 2369–2382, Aug. 2019, doi: 10.1109/TNNLS.2018.2884700.

[122] M. Chalupa, "Slicing of LLVM bitcode," Masaryk University, Brno, CZ, 2016. [Online]. Available: https://is.muni.cz/th/vik1f/thesis.pdf

[123] Y. Sui, "Developing practical pointer analysis for large-scale software," UNSW Sydney, 2014. doi: 10.26190/UNSWORKS/16925.

[124] P. D. Schubert, B. Hermann, and E. Bodden, "PhASAR: An Inter-procedural Static Analysis Framework for C/C++," in *Tools and Algorithms for the Construction and Analysis of Systems*, Cham, 2019, pp. 393–410.

[125] Y. Sui and J. Xue, "SVF: interprocedural static value-flow analysis in LLVM," in *Proceedings of the 25th International Conference on Compiler Construction*, New York, NY, USA, Mar. 2016, pp. 265–266. doi: 10.1145/2892208.2892235.

[126] Y. Sui and J. Xue, "Value-Flow-Based Demand-Driven Pointer Analysis for C and C++," *IIEEE Trans. Software Eng.*, vol. 46, no. 8, pp. 812–835, Aug. 2020, doi: 10.1109/TSE.2018.2869336.

[127] N. Francis *et al.*, "Cypher: An Evolving Query Language for Property Graphs," in *Proceedings of the 2018 International Conference on Management of Data*, Houston TX USA, May 2018, pp. 1433–1445. doi: 10.1145/3183713.3190657.

[128] Docker Inc., "The Industry-Leading Container Runtime - Docker," Sep. 28, 2021. https://www.docker.com/products/container-runtime/ (accessed Oct. 12, 2022).

[129] 2022 Neo4j, Inc., "Extending Neo4j - Java Reference," *Neo4j Graph Data Platform*, 2022. https://neo4j.com/docs/java-reference/4.4/extending-neo4j/ (accessed Oct. 13, 2022).

[130] L. McInnes, J. Healy, and J. Melville, "UMAP: Uniform Manifold Approximation and Projection for Dimension Reduction." arXiv, Sep. 17, 2020. Accessed: Nov. 17, 2022. [Online]. Available: http://arxiv.org/abs/1802.03426

[131] D. Chicco and G. Jurman, "The advantages of the Matthews correlation coefficient (MCC) over F1 score and accuracy in binary classification evaluation," *BMC Genomics*, vol. 21, no. 1, p. 6, Jan. 2020, doi: 10/gg2rv3.

[132] J. Fan, Y. Li, S. Wang, and T. N. Nguyen, "A C/C++ Code Vulnerability Dataset with Code Changes and CVE Summaries," in *Proceedings of the 17th International Conference on Mining Software Repositories*, Seoul Republic of Korea, Jun. 2020, pp. 508–512. doi: 10.1145/3379597.3387501.

[133] D. J. Pearce and D. J. Pearce, "An Improved Algorithm for Finding the Strongly Connected Components of a Directed Graph," 2005.

[134] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu, "A density-based algorithm for discovering clusters in large spatial databases with noise," in *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, Portland, Oregon, Aug. 1996, pp. 226–231.

[135] G. Montavon, A. Binder, S. Lapuschkin, W. Samek, and K.-R. Müller, "Layer-Wise Relevance Propagation: An Overview," in *Explainable AI: Interpreting, Explaining and Visualizing Deep Learning*, W. Samek, G. Montavon, A. Vedaldi, L. K. Hansen, and K.-R. Müller, Eds. Cham: Springer International Publishing, 2019, pp. 193–209. doi: 10.1007/978-3-030-28954-6_10; https://web.archive.org/web/20210825130349/https://link.springer.com/chapter/10.1007%2F978-3-030-28954-6_10.

[136] The OWASP Foundation Inc., "OWASP Foundation | Open Source Foundation for Application Security," 2021. https://owasp.org/ (accessed Nov. 10, 2021).

[137] S. Chakraborty, R. Krishna, Y. Ding, and B. Ray, "Deep Learning based Vulnerability Detection: Are We There Yet," *IIEEE Trans. Software Eng.*, pp. 1–1, 2021, doi: 10/gk52qr.

[138] M. Weiser, "Program slicing," in *Proceedings of the 5th international conference on Software engineering*, San Diego, California, USA, Mar. 1981, pp. 439–449.