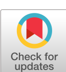


Combining Structured Static Code Information and Dynamic Symbolic Traces for Software Vulnerability Prediction

Huanting Wang
Northwest University, China
University of Leeds, U. K.

Zhanyong Tang*
Northwest University, China

Shin Hwei Tan
Concordia University, Canada

Jie Wang
Northwest University, China

Yuzhe Liu
Northwest University, China

Hejun Fang
Northwest University, China

Chunwei Xia
University of Leeds, U. K.

Zheng Wang*
University of Leeds, U. K.

ABSTRACT

Deep learning (DL) has emerged as a viable means for identifying software bugs and vulnerabilities. The success of DL relies on having a suitable representation of the problem domain. However, existing DL-based solutions for learning program representations have limitations – they either cannot capture the deep, precise program semantics or suffer from poor scalability. We present Concoction, the first DL system to learn program presentations by combining static source code information and dynamic program execution traces. Concoction employs unsupervised active learning techniques to determine a subset of important paths to collect dynamic symbolic execution traces. By implementing a focused symbolic execution solution, Concoction brings the benefits of static and dynamic code features while reducing the expensive symbolic execution overhead. We integrate Concoction with fuzzing techniques to detect function-level code vulnerabilities in C programs from 20 open-source projects. In 200 hours of automated concurrent test runs, Concoction has successfully uncovered vulnerabilities in all tested projects, identifying 54 unique vulnerabilities and yielding 37 new, unique CVE IDs. Concoction also significantly outperforms 16 prior methods by providing higher accuracy and lower false positive rates.

CCS CONCEPTS

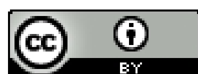
• **Security and privacy** → **Software security engineering**; • **Computing methodologies** → *Artificial intelligence.*

KEYWORDS

Software vulnerability detection, Deep learning, Symbolic execution

ACM Reference Format:
Huanting Wang, Zhanyong Tang*, Shin Hwei Tan, Jie Wang, Yuzhe Liu, Hejun Fang, Chunwei Xia, and Zheng Wang. 2024. Combining Structured Static Code Information and Dynamic Symbolic Traces for Software Vulnerability Prediction. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3597503.3639212

*Corresponding faculty authors: Zhanyong Tang and Zheng Wang.



This work is licensed under a Creative Commons Attribution International 4.0 License.

ICSE '24, April 14–20, 2024, Lisbon, Portugal

© 2024 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-0217-4/24/04.
https://doi.org/10.1145/3597503.3639212

1 INTRODUCTION

Despite significant efforts to enhance software reliability, software vulnerabilities remain a primary concern in modern software development [40, 58]. In recent years, deep learning (DL) techniques have emerged as a powerful method for constructing sophisticated tools and models to identify common software bugs and vulnerabilities [21, 57, 59, 60, 63, 80, 84]. By training a predictive model on large volumes of training data, DL models can learn the latent code patterns indicative of vulnerable programs. Once trained, these models can be applied to new, previously unseen programs to identify potentially buggy code [8, 58].

The success of a supervised learning model is heavily dependent on having a suitable representation of the problem domain that can encode the essential information needed for the task at hand, such as vulnerability detection in our case [72]. In the context of DL-based code modeling, this requires constructing numerical vectors, or *embeddings*, that capture the important characteristics of the program source code or binary.

The vast majority of DL-based code modeling techniques rely on deep neural networks (DNNs) to learn program representations from static code, such as source code texts [57, 59, 60], abstract syntax trees (ASTs) [54, 75], program data and control flow graphs (PDCGs) [21, 63, 88], or a combination of these [80]. While static code information can capture all possible program execution paths, it can suffer from complex and ambiguous information due to redundant statements, complex data structures, and extensive execution paths in the source code. Like classical compiler analysis, this can lead to over-conservative decisions and a high false-positive rate[1], and a low true positive ratio [15] for automatic bug detection.

More recent approaches, like LIGER [81], attempted to use dynamic execution traces to learn program representation. These approaches utilize execution statements seen during profiling to represent static program information and track changes in program

[1]In this paper, a *false positive* occurs when the code does not contain a bug or vulnerability, but the detection model indicates otherwise. By contrast, a *false negative* occurs when the model fails to identify a true bug or vulnerability.

variables to capture dynamic program behavior. By considering dynamic execution paths, symbolic traces provide precise information about dynamic program behavior and reduce false-positive rates in code analysis. While promising, prior approaches have two limitations. Firstly, they solely rely on executed code statements seen for static code representation. This can suffer from poor coverage and overlook the structured data flow and dependence information available in the static program graph. This comprehensive data and control flow information is crucial for vulnerability detection, encompassing all possible execution paths. Secondly, they employ random sampling for dynamic tracing, which presents challenges when applying dynamic tracing methods to real-world software projects due to the expensive overhead of symbolic executions [18].

In this paper, we ask the question, "*what if we could bring the best of static and dynamic code information in a single DL framework for code vulnerability detection?*". In response, we develop Concoction, a new DL system to combine static and dynamic code representation to detect software bugs and vulnerabilities at the source code level. Specifically, static code information, such as PDCG (Program Data and Control-flow dependence Graph), offers a high-level view of all possible program behaviors and data flow information, which can mitigate the coverage issue of symbolic execution. On the other hand, symbolic execution traces on *a small set* of carefully selected execution paths can provide more precise information and deeper program semantics to disambiguate static code information. By integrating these two types of information, we avoid the computational overhead of running symbolic executions on every possible execution path while still leveraging the benefits of deep program semantics provided by dynamic executions.

Concoction utilizes a Transformer-based DNN architecture [78] to leverage static and dynamic code information. The model comprises a *representation component* and a *detection component*. The representation component maps static and dynamic program information into a joint embedding (or feature vector) for program representations. The detection component takes the program representation as input and predicts whether the input code contains a vulnerability. One of the key features of Concoction is its ability to minimize the overhead of dynamic tracing. This is achieved by employing a path selection component during deployment to determine which execution paths from the PDCG should be chosen to collect symbolic execution traces. These traces are then fed into the representation component to generate dynamic embeddings, which are combined with the static embeddings and passed to the detection model for vulnerability prediction.

To overcome the challenge of limited numbers of labeled code samples, Concoction combines supervised and unsupervised learning techniques. We first leverage unsupervised contrastive learning to pretrain the representation network. For this purpose, we employ the language masking method [76] and train the representation model on a dataset of 100K unlabeled C functions sourced from GitHub and open datasets. To generate additional training samples, we introduce a dropout-based contrastive learning component [34]. The contrastive loss function encourages the model to understand code semantics better by mapping similar samples closely and distantiating those with different semantics in the embedding space. Furthermore, we also extend unsupervised learning to train the path selection component using the same unlabeled training dataset. Once the representation component is trained, we remove the contrastive learning network and combine the representation layers with the detection component, creating an end-to-end model. This model takes joint embeddings of static and dynamic information as input and predicts whether the input code contains a bug. To train the end-to-end model, we use the learned weights of the representation model to initialize the corresponding layers of the model and fine-tune the entire architecture on a dataset of 14K labeled code samples obtained from public datasets, including the Software Assurance Reference Dataset (SARD) [64] and Common Vulnerabilities and Exposures (CVE) [2].

We have implemented a working prototype of Concoction[2]. Our implementation utilizes KLEE [17] to generate the symbolic execution traces. We demonstrate the benefits of Concoction by applying it to C programs to detect function-level vulnerabilities from source code. We further integrate Concoction with fuzzing test techniques [12] to automatically generate bug-exposing test cases when a function is predicted to have a code vulnerability, aiming to minimize the effort of manual examination.

We evaluate Concoction by applying it to 20 diverse, real-life open-source projects that are not presented in our training dataset. We compare Concoction against 16 prior methods, including eleven state-of-the-art learning-based methods [21, 23, 33, 39, 43, 57, 60, 63, 80, 81, 88], two symbolic execution engines [16, 17], two static analysis tools [1, 4] and a fuzzing tool [31] for identifying security flaws. Experimental results show that Concoction consistently outperforms 16 competing methods across evaluation settings by discovering more code vulnerabilities with a lower false-positive rate. In less than 200 hours of automated concurrent testing runs, Concoction has uncovered vulnerabilities in all tested projects and successfully identified 54 software vulnerabilities, with 37 new CVE IDs assigned.

This paper makes the following contributions:

- It presents the first DL framework to effectively combine structured code information and symbolic executions for vulnerability prediction (Sec. 3.2);
- It demonstrates how unsupervised active learning can be employed to reduce symbolic execution overhead through execution path selection (Sec. 3.4);
- The release of two open datasets to facilitate research in leveraging static code information and symbolic execution traces.

2 MOTIVATION

Static code analysis techniques for bug detection can suffer from false positives (incorrectly flagging a bug). For example, Figure 1 shows a function `b()` which will not be invoked during execution because the static array `a` is initialized to zero by definition, and the condition at line 9 is evaluated to false. However, modern compilers such as GCC and LLVM cannot detect dead code due to complex and inaccurate pointer aliasing analysis. Similarly, DL-based bug detection approaches [21, 39, 57, 59, 60, 63, 80, 88] that rely on static code features predict that this example contains a "*division-by-zero*" vulnerability at line 4, which is a false positive.

Avoiding a *false positive* (FP) of Figure 1 would require capturing program semantics between variables, function calls and structured

[2] Code and data are available at https://github.com/HuantWang/CONCOCTION.

```
1  extern void foo(void);
2  static int a[] = {0,0};
3  static int b(int c)
4  { return a[1] % c; } // false positive
5
6  int test() {
7      static int *a_ptr, c;
8      a_ptr = &(a[1]);
9      if (*a_ptr) { // unreachable branch
10         if (b(c)) {foo();}
11     }
12     return 0;}
```

Figure 1: This example contains a false "*division-by-zero*" issue at line 4 because the branch at line 10 will not be taken.

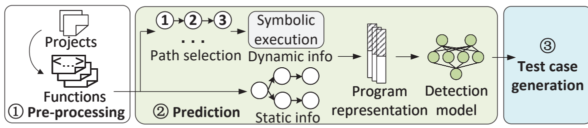


Figure 2: Workflow of Concoction during deployment.

data. An approach that uses only static information may not accurately trace the data flow across function calls and data structures. Meanwhile, DL-based solutions based on static code information, such as AST or PDCG, suffer from the same issue.

Can we do better by combining static and dynamic information? This is the insight shared in this paper. For this example, we can infer from symbolic execution traces that the function `b()` will not be executed due to the values in array a by inlining the callee function `b()` to `test()`. The static PDCG further reveals that `a` is an invariant in all possible execution paths. Combining static and dynamic information, we can observe that this branch is never taken, making a "*division-by-zero*" error impossible.

A natural question is: "*why not just rely on symbolic executions?*". In an ideal world where computation resources and symbolic execution overhead are not an issue, a symbolic execution engine will be able to identify the vulnerability of this example through exhaustive executions. However, this is often infeasible because exhaustively trying all possible execution paths is prohibitively expensive. This example highlights the need to leverage static and dynamic program information for code vulnerability detection. Concoction is designed to offer this capability.

3 OUR APPROACH

Concoction is a DL framework for detecting software vulnerabilities in source code. In this work, we apply Concoction to identify bugs at the function level in C programs. Specifically, our work focuses on detecting bugs and security flaws defined in the CWE database [3]. In practice, Concoction can be integrated into an automated build system like Jenkins [5] to execute the vulnerability detection process as a background process when a new merge request is submitted. As these build systems often run overnight on a dedicated backend server, they do not affect the standard development activities and the overhead of Concoction should be acceptable to many developers.

The *key technical contributions* of Concoction include: (1) combining structured static code information and symbolic execution traces to learn program representation (Sec. 3.2.1 and 3.2.2), and (2) a learnable path selection component to reduce symbolic execution overhead (Sec. 3.4). Concoction builds upon prior foundations in enhanced AST (Sec. 3.2.1), Transformer-based neural architectures, and contrastive learning (Sec. 3.3).

3.1 Overview of Concoction

Figure 2 depicts the workflow of using Concoction to detect function-level code vulnerabilities *during deployment*.

Pre-processing. Concoction uses a LLVM compiler plugin [50] to partition the project code into individual functions by inlining callee functions, relevant data structures, and global variables.

Prediction. Concoction extracts two types of information for each target function: (1) AST and PDCG from static source code, and (2) the symbolic execution traces of selected execution paths using a symbolic execution engine [17]. It employs a path selection component (Sec. 3.4) to identify critical paths to collect dynamic symbolic execution traces. The static and dynamic information produce static and dynamic embeddings through dedicated representation networks, which are then concatenated to create a joint representation to be used by the detection model for prediction.

Test case generation. When the model detects a potential vulnerability in the input function, it invokes a fuzzing engine to identify and expose the weakness by generating randomized test inputs for the function. As we only employ fuzzing for functions suspected to contain vulnerabilities, the fuzzing overhead is manageable, taking less than 12 hours for all fuzzed functions within a project.

3.2 The Concoction Architecture

Figure 3 shows the workflow of training the Concoction DL components for program representations and vulnerability detection.

Program representation. Our representation component uses two Transformer-based networks to map the input source code and symbolic execution traces into a numerical embedding vector. Then, a dense layer concatenates the embeddings generated by the two networks to a joint vector as the output. We set the embedding length of the static and dynamic embedding vectors to 100 dimensions, leading to a joint embedding vector of 200 dimensions. As in prior work [37], using a larger dimension does not yield better performance in our setting but may increase the training overhead.

Vulnerability detection. The detection component is a multi-layer perceptron (MLP) network that takes joint embedding to predict the vulnerability. It includes a fully connected layer, a dropout layer with a rate of 0.1, and a sigmoid layer. Our current implementation only predicts if a function may contain a vulnerability and does not identify the type of vulnerability.

3.2.1 *Extracting static code information.*

We use a parser built upon the Language Server Protocol (LSP) [38] to rewrite the variable names with a consistent naming scheme. This step handles syntactic variations in the programs. Next, we construct an enhanced AST, using the LSP, which contains the standard *syntax nodes*, i.e., nonterminals in the language grammar like an AST node for an `if` statement or function declaration, and *syntax tokens*, i.e., terminals like identifier names and constant values. Following [9], we also

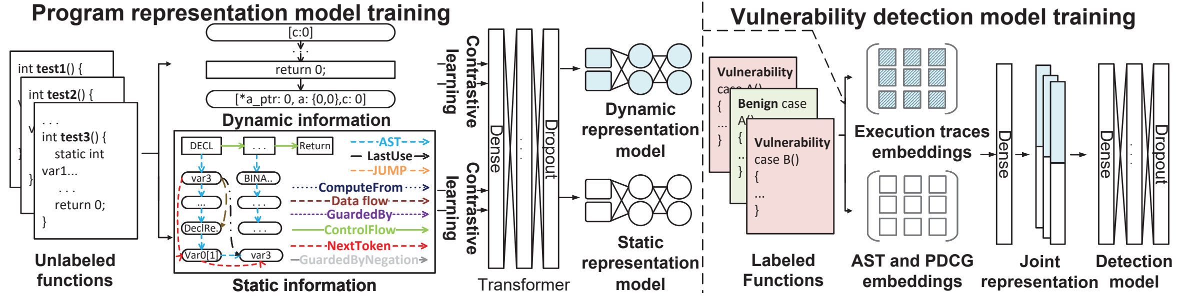


Figure 3: The CONCOCTION DNN architecture and its training workflow.

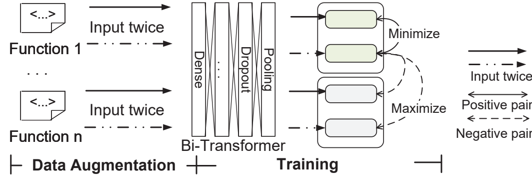


Figure 4: Contrastive learning for representation training.

introduce eight additional types of edges to the AST, including *Child*, *Data and Control Flows*, *GuardedBy*, *Jump*, *ComputedFrom*, *NextToken* and *LastUse and LastLexicalUse*, shown in Figure 3.

3.2.2 Extracting dynamic information. We use KLEE [17] to obtain symbolic execution traces. To pre-train the representation model (Sec. 3.3), we generate execution traces using a non-uniform random search heuristic to explore different execution paths. During deployment, symbolic traces are generated solely on the selected paths instead of random search to minimize the overhead of symbolic executions (Sec. 3.4). We terminate symbolic execution after a configurable time limit (4 hours for collecting training data and 5 minutes when using the trained model). Subsequently, we combine different symbolic inputs and their corresponding reachable program paths as a sequence of execution traces to be fed into the dynamic embedding network (Figure 3).

3.3 Contrastive Pre-training

We use a bidirectional Transformer network [39] to learn static and dynamic program embeddings. We pre-train the static and dynamic embedding models separately on the same unlabeled dataset. Our pre-training dataset contains 100K C code snippets collected from GitHub and SARD shown as Table 1. After training, we use the output of the last hidden layer of the embedding networks as the static or dynamic embedding vector. We employ contrastive learning to increase the dataset size and enhance the model's robustness.

3.3.1 Model inputs. We pair the source code text and flattened enhanced AST sequence, sending them to the static embedding network. The AST sequence is generated by traversing the AST in a breadth-first manner. During training, the static embedding model predicts masked tokens from either the source code or AST's data and control flow relations to generate contextual representations. Likewise, the dynamic embedding network takes symbolic execution traces and maps them to an embedding vector, capturing temporal dependencies and runtime behavior of the program.

3.3.2 Training methodology. To enhance the model's robustness and generalization ability, we employ dropout-based contrastive learning [34]. Our approach includes dense, dropout, and pooling layers, depicted in Figure 4. Specifically, we individually attach the contrastive learning component to the static and dynamic representation networks. Then, we train each combined network separately and remove them from the contrastive learning component.

Contrastive learning increases the training dataset size by adding noises to the data. CONCOCTION randomly disables neurons in the representation network, generating various dropouts as shown in Figure 4. Specifically, it passes the code sample inputs x_i through the representation network twice with different dropout probabilities, resulting in two different embeddings as a "positive pair" for x_i. Another sample is paired with x_i to create a "negative pair". The contrastive learning component is then used to predict positive samples from negatives in a training mini-batch and calculate the loss. The training process aims to minimize the standard Noise Contrastive Estimate (NCE) loss function [47] to maximize the agreement between semantically similar pairs.

3.3.3 Training the end-to-end detection model. After training the static and dynamic embedding networks, we attach them to a dense layer to create joint embeddings for the detection network. This forms the final end-to-end architecture shown in the right part of Figure 3. The joint representation component serves as the encoder, and we initialize its weights using those obtained during pre-training. We train the end-to-end network using *labeled* data samples, which consist of 13,768 C code snippets from CVE and SARD datasets. Each sample has a two-dimensional one-hot label indicating whether it contains a vulnerability. Vulnerable code samples are collected from open-source projects using the assigned CVE or SARD IDs, whereas benign samples are obtained from the patched version of the same project. For each training sample, we generate an enhanced AST and randomly sampled symbolic traces (Sec. 3.2.1 and 3.2.2). We use the *pre-trained* representation component to generate the joint embedding as the program representation, which becomes the detection model's input. Our end-to-end model is trained to optimize the cross-entropy loss for classifications.

3.4 Path Selection for Symbolic Execution

After training the end-to-end model, we use the path selection component to choose significant paths for symbolic executions *during deployment.* This differs from execution traces collected during training, where we use a random sampling scheme to improve the

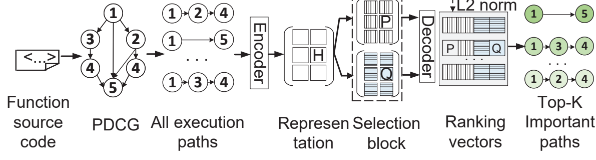


Figure 5: Rank and select paths for symbolic execution.

training data size, as symbolic execution overhead is less of an issue for offline model training.

3.4.1 Overview of path selection. Figure 5 shows the workflow of choosing k most important paths from all possible execution paths of the target function. Like ContraFlow [23], we use unsupervised active learning to identify most representative paths for encoding programs [51], such that the number of paths for code embedding is reduced while important program semantics are well-preserved. Unlike ContraFlow, our goal is to select paths to collect symbolic execution traces. As we will show in Sec. 5.2 and 5.3, our approach outperforms ContraFlow.

Our goal. Given n execution paths $\boldsymbol{H} = [h_1, ..., h_n]$ collected from the PDCG of the test sample, our goal is to choose a subset of important paths to collect symbolic execution traces, which h_i is an embedding vector generated by our static representation model. The objective is to choose representative data points that can reconstruct most of the input PDCG information and preserve the unique features of the sample. In this work, we use the K-means clustering algorithm to model the path features by grouping data points into clusters on the embedding space. The path selection component is trained to find a sample subset that captures the input patterns and preserves the cluster structure of the data.

3.4.2 Execution path representation. For each test sample (function), we extract static execution paths from the PDCG. For each static execution path, we first eliminate irrelevant code (and AST nodes) and then feed the code segmentation to the trained static representation model (fine-tuned when training the end-to-end model) to generate embeddings for the code segments. This process is repeated for each static path, resulting in a matrix, $\boldsymbol{H}$, to serve as the input of the path selection component, which each matrix element being an embedding vector produced by our static representation model. Since path collection involves traversing the PDCG without program execution, the overhead is negligible.

3.4.3 Network structure. As our path selection component uses unsupervised learning (we do not have labels to tag if a path is important), using an encoder-decoder network to map the input into a latent space for path selection is a natural choice. We add a *selection block* between the encoder and decoder to select samples (or paths) from all input paths to be passed into the decoder. As the selection block is differentiable, the selection network is trainable.

Selection block. The selection block comprises two branches, each consisting of a single fully connected layer without bias and nonlinear activation functions. The first branch aims to identify a subset of paths ($\boldsymbol{H'}$) that can effectively approximate all paths from the input matrix ($\boldsymbol{H}$). The second branch initially clusters the data in the latent space and then selects a sample subset approximating the resulting cluster centroids.

3.4.4 Unsupervised active learning. For all input paths of a test sample, our approach constructs two coefficient matrices, $\boldsymbol{Q}$ and $\boldsymbol{P}$, where each matrix element is a d-dimensional embedding vector produced by the encoder. The matrix $\boldsymbol{Q}$ is constructed by the first branch of the selection block to approximate the input matrix ($\boldsymbol{H}$), while the matrix $\boldsymbol{P}$, constructed by the second branch of the selection block, aims to preserve the cluster structure when applying K-means to the latent space learned by the encoder.

Training objectives. Our active learning process refines the matrices to maximize the distance between (1) $\boldsymbol{H}$ and matrix $\boldsymbol{Q}$, (2) the cluster centroid matrix $\boldsymbol{C}$ obtained using K-means clustering on $\boldsymbol{H}$, and the reconstructed matrix $\boldsymbol{P}$, and (3) $\boldsymbol{H}$ and the decoder outputs.

Loss function. Our overall loss function is defined as $Min\ \ell = \alpha\ell_a + \beta\ell_b + \ell_c$, where α and β are tradeoff parameters. The terms in the loss function are:

$$\ell_a = ||\phi(H) - \phi(H)Q||_F^2 + \gamma||Q||_2 \quad (1)$$

$$\ell_b = ||C - \phi(H)P||_F^2 + \eta||P||_2 \quad (2)$$

$$\ell_c = ||\phi(H) - G||_F^2 \quad (3)$$

Eq. 1 approximates the input patterns. The value of ε represents either 2 (l2-norms) or F (Frobenius norm) for the normalization function $||\cdot||_\varepsilon$, which is used to measure the informativeness of each feature. $\phi(H)$ is a nonlinear transformation that maps input paths H to a new latent representation, and the tradeoff parameter γ controls the balance between the reconstruction loss and the regularization term. Eq. 2 aims to minimize the cluster centroids reconstruction loss, corresponding to objective (2), with the tradeoff parameter η. Finally, Eq. 3 corresponds to the reconstruction loss of the encoder-decoder model, which represents the third objective. $\boldsymbol{G}$ denotes the decoder's output for a given input $\phi(H)$.

Training process. We iteratively train the selection component on the unlabeled Concoction training data. Firstly, we pre-train the encoder and decoder without considering the selection block. After that, we perform K-means on the encoder output and consider the obtained K cluster centroids as the centroid matrix $\boldsymbol{C}$ for subsequent sample selection. The number of clusters (K) is determined automatically using the Bayesian information criterion (BIC) [62]. Finally, we use the pre-trained parameters to initialize the encoder and decoder, batch all data, and minimize the overall loss function using Adam optimizer with a 0.001 learning rate.

3.4.5 Path selection during deployment. Once the selection component is trained on the Concoction training data, we can obtain two reconstruction coefficient matrices $\boldsymbol{Q}$ and $\boldsymbol{P}$. To use the selection component, we normalize the columns of $\boldsymbol{Q}$ and $\boldsymbol{P}$ using l2-norm and convert the values to $[0, 1]$. This produces two ranking vectors $\hat{q}, \hat{p} \in \mathbb{R}^n$, which we merge and sort in descending order to identify K top-ranked paths. Parameter K can be flexibly set by the user. In this work, we set K to be 30%, sufficient to cover important paths of vulnerable functions in our *training* dataset. If the number of paths of the target code is less than 10, we consider all execution paths as the overhead of symbolic executions is small.

3.4.6 Symbolic execution for chosen paths. We extended KLEE [17] to cover the selected paths of the target function. To do so, we use a compiler-based pass to insert a callback function to the target program to guide KLEE skip paths not selected by our path selection

Table 1: Open datasets used in training and evaluation

Source	#Projects	Versions	#Samples	#Vulun. samples
SARD	/	/	30,954	5,477
CVE	Jasper	v1.900.1-5, v2.0.12	24,330	663
	Libtiff	v4.0.3-9	4,896	558
	Libzip	v0.10, v1.2.0	5,618	49
	Libyaml	v0.1.4	27,773	41
	SQLite	v3.8.2	1,794	31
	ok-file-fromats	203defd	1,014	25
	libpng	v1.2.7, v1.5.4, v1.6.0	954	12
	libming	v0.4.7-8	1,104	15
	libexpat	v2.0.1	1,051	13

component. We also use a script to record the addresses, sizes, and names of all variables of the target function during symbolic execution. The script also generates a test driver program for KLEE to facilitate the execution of KLEE.

By skipping unwanted execution traces early, we can manage the overhead of symbolic execution effectively. We terminate symbolic execution after a configurable threshold (5 minutes in this work) during evaluation. The symbolic inputs and corresponding reachable code paths produce execution traces that we pass to the dynamic representation model to generate dynamic embeddings. Note that our approach guarantees that there are always symbolic execution traces generated for the test function.

3.4.7 Targeted fuzzing for test case generation. When our detection model identifies a potentially buggy function that symbolic execution fails to expose, we use AFL++ [31] (an AFL extension) for fuzzing functions predicted to be vulnerable. The idea is to generate bug-exposing test cases to help developers analyze and verify the identified vulnerabilities. In this work, we use the AFL++ partial instrumentation mode to guide AFL++ towards targeting functions predicted by Concoction to have vulnerabilities. To this end, we provide AFL++ with the project's source code, build scripts, vulnerable functions identified by Concoction, and seed program inputs to be mutated using the default AFL++ configurations. We then ask AFL++ to instrument the compiled assembly code and mutate the seed test inputs to cover the target functions. Fuzzing is terminated if AFL++ detects a program crash or exceeds a 12-hour runtime (which may involve fuzzing multiple functions together). We then manually verified and reported the issue to developers with information to reproduce the issue. If AFL++ does not trigger any crash, we manually examine the predicted function to check for a vulnerability and file an issue report for each confirmed bug.

4 EXPERIMENTAL SETUP

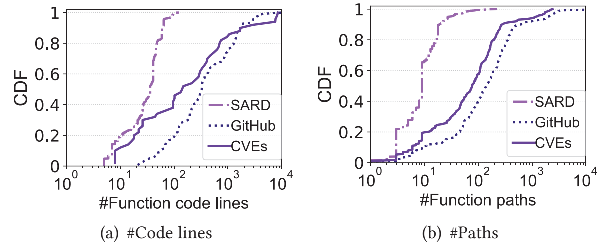
We implemented Concoction in 10+K lines of Python and 5K lines of C/C++ code. Our DL model is implemented using PyTorch (ver. 1.13). We use Joern (ver. 1.1) [6] to construct the AST for static code representation and KLEE (ver. 2.1) to collect the symbolic execution traces. We train and test Concoction models and all baselines on a multi-core server with two 32-core AMD EPYC 7532 CPUs at 2.40GHz and an NVIDIA 2080Ti GPU. The server has 128GB of RAM and runs Ubuntu 18.04 with Linux kernel ver. 5.4.

4.1 Workloads

Open-source projects. We applied Concoction to 20 open-source projects from various domains. These projects, listed in Table 3, were chosen because they are widely used, have been used in related work [7], or have active development teams. We stress that none of these projects was used to train Concoction, and we tested the latest version of each project at the time of testing. We also note that bugs discovered by Concoction were previously unreported at the time of testing; hence, there were no data leakage issues.

Table 2: Open-source projects with known CVEs

No.	Project	Versions	#Lines of code	#vuln.
1	SQLite	v3.30.1, v3.8.2	242K	13
2	Libtiff	v4.0.9	140K	10
3	Libpng	v1.2.7, v1.5.4	32K	12



(a) #Code lines (b) #Paths

Figure 6: CDF for the (log-scale) number of lines (a) and execution paths (b) our test samples.

Open datasets. In Sec. 5.2 and 5.3, we compare Concoction with prior work on three datasets used by the prior work for evaluation purposes. In Sec. 5.2.1 and 5.2.2, we use samples from SARD [64] and CVE datasets (see Table 1), respectively. In Sec. 5.3, we evaluate Concoction on three open-source projects that have known CVEs (see Table 2). We use three-fold cross-validation to evaluate all approaches on the above datasets, where samples are split on the project-level, and samples of a test project are excluded from the training dataset.

Data collection and workload characteristics. Our program representation model, Concoction, was trained and tested on a dataset of over 100K functions from SARD and 9 large C-language open-source projects (Table 1). Four security researchers spent 600 man-hours on manual labeling and cross-verification to collect these samples. Additionally, we spent 200+ machine hours extracting dynamic and static information using KLEE. Figure 6 shows the cumulative distribution functions (CDF) of the number of lines and execution paths in the test samples in Tables 1 and 3. The SARD dataset consists mainly of short functions, where over 50% have <40 lines of code and 4 paths, leading to high detection accuracy with baseline methods. However, the functions from the CVE dataset and open-source projects are much larger, with over 50% containing $\geq$ 400 lines of code (up to 10K) and 128 paths (up to 12K). This increased complexity in the CVE dataset and open-source projects reduces the accuracy and recall for our baselines compared to SARD.

4.2 Competing Baselines

We evaluate Concoction by comparing it with 16 prior methods. These include (1) eleven state-of-the-art DL-based models, (2) two symbolic execution tools, (3) one fuzzing tool that our approach relies on, and (4) two static analysis tools. Before running the DL baselines on the same datasets, we ensure that our evaluation setup

achieves results comparable to those reported in their source publications for a fair comparison.

DL models based on static information. We compared Concoction against eleven DL models that use static code information. These include Vuldeepecker [57], which utilizes a BiLSTM architecture, as well as Funded [80], Devign [88], ReVeal [21], and ReGVD [63], which employ a variant of graph neural networks to learn program representations. We also compared against LineVul [33], LineVD [43], CodeXGLUE [60] and GraphcodeBERT [39], which use the Transformer architecture, and ContraFlow [23] which utilizes contrastive learning to represent the code, followed by an LSTM architecture to identify vulnerabilities.

DL models based on dynamic information. LIGER [81] is a closely related work that learns program representations from symbolic execution traces. However, unlike our approach, LIGER uses a random sampling method for collecting symbolic traces and does not utilize structured data flow and dependence information from static source code. Additionally, it employs a multi-tier recurrent neural network (RNN) architecture, whereas Concoction uses the Transformer architecture. For our evaluation, we used the open-source implementation of LIGER, adapting it for vulnerability detection and training it on the same datasets as Concoction.

Static tools. We compare Concoction with two representative static analysis tools: CodeQL [1] and Infer [4], by using the default, recommended configurations of the tools.

Symbolic execution engines. We also compare Concoction to two state-of-the-art symbolic execution tools, including KLEE [17] and MoKLEE [16]. The latter is designed to reduce the overhead of symbolic executions by allowing symbolic executions to run on previous paths while continuing to explore new paths.

Fuzzing tool. As noted in Sec. 3.4.7, Concoction uses AFL++ to fuzz predicted buggy functions and generate bug-exposing tests. We compare Concoction with the native AFL++, which tests the entire program without Concoction's guidance. For a fair comparison, both AFL++ and Concoction use the same seed program inputs.

4.3 Evaluation Methodology

We applied Concoction to 20 open-source projects and 14K function-level source code samples (vulnerable and benign). Our evaluation is designed to answer the following research questions:

RQ1: Does combining static and dynamic information help detect code vulnerabilities in real-life open-source projects (Sec 5.1)?
RQ2: How does Concoction compare with prior approaches in detecting function-level vulnerabilities (Sec. 5.2 and 5.3)?
RQ3: How do individual components of Concoction contribute to its overall performance (Sec. 5.4)?

Evaluation metrics. We consider four *higher-is-better* statistical metrics: *accuracy*, *precision*, *recall* and the *F1 score*. Accuracy is computed as the ratio of correctly labeled cases to the total test cases. Precision is the ratio of correctly predicted samples to the total number of samples predicted to have the same label. It answers the question, "*Out of all the samples predicted to contain a vulnerability, how many are correct?*" High precision indicates a low *false-positive* rate, meaning that a lower proportion of the samples predicted to

Table 3: Vulnerability statistics for each tested project.

Projects	Versions	Release date	#Stars	#Lines of code	#Sub-mitted	#Confirmed	
						#Verified	#Fixed in verf.
Linux Kernel	v6.1-rc5, v6.1-rc4	Apr. 2023	160k	30M	2	2	0
assimp	v5.1.4	Sep. 2023	9.7k	374k	6	6 (6 CVE)	0
Image-Magick	v7.0.11-5	Jul. 2023	10.2k	42k	1	1 (1 CVE)	1
lepton	v1.0-1.2.1	Feb. 2023	5k	80k	1	1 (1 CVE)	1
zydis	770c320	Apr. 2023	3.0k	80k	4	4	4
openEXR	v2.2.0	Jul. 2023	1.5k	246k	1	1 (1 CVE)	1
openjpeg	a44547d	Apr. 2023	902	124k	1	1 (1 CVE)	1
Leanify	b5f2efc	Dec. 2022	801	61k	2	2	2
astc-encoder	v3.2k	Jun. 2023	885	148k	3	3 (2 CVEs)	3
AudioFile	004065d	Apr. 2023	355	7k	1	1 (1 CVE)	1
xlsxio	af485eb	Nov. 2022	231	9k	1	1	1
mediancut-posterizer	v2.1	Feb. 2023	203	1.8k	1	1 (1 CVE)	0
ELFLoader	34fd7ba	May 2022	203	3k	4	4	0
pdftojson	94204bb	Oct. 2017	138	148k	3	2 (2 CVEs)	0
epub2txt2	71dc41	Jun. 2022	153	10k	1	1 (1 CVE)	1
deark	v1.6.2	Jul. 2023	136	154k	3	3 (1 CVE)	3
ok-file-formats	203defd	Sep. 2021	136	15k	7	7 (7 CVEs)	7
sqlcheck	391ae84	Mar. 2022	2.3k	4.5k	4	4 (4 CVEs)	0
packJPG	v2.5k	Apr. 2020	151	11k	7	7 (7 CVEs)	0
json2xml	v3.14.0	Nov. 2023	88	2.9k	1	1 (1 CVE)	1
Total	/	/	/	/	**54**	**53 (37 CVEs)**	**27**

Table 4: Top-3 types of issues found by Concoction.

Category	#Submitted	#Confirmed	#Fixed	#Dyn-related
buffer-overflow	33	33	20	23
segmentation-violation	6	6	1	5
memory-leaks	4	3	1	3
other types	11	11	5	6
Total	**54**	**53**	**27**	**37**

have bugs are bug-free. Recall is the ratio of correctly predicted samples to the total number of test samples belonging to a class. It answers questions like "*Of all the vulnerable test samples, how many are actually predicted to be vulnerable?*". High recall suggests a low *false-negative* rate. Finally, the F1 score is the harmonic mean of Precision and Recall, calculated as $2 \times \frac{Recall \times Precision}{Recall + Precision}$. It is useful when the test data have an uneven label distribution.

5 EXPERIMENTAL RESULTS

5.1 Detect Vulnerabilities in Large-scale Testing

This subsection quantifies Concoction's ability to detect function-level code vulnerabilities in the 20 projects listed in Table 3. For ethical considerations, we first contacted the developers through a private email for vulnerabilities that are likely exploitable (including all those with a CVE ID assigned), and followed their advice.

5.1.1 Vulnerability count. Table 3 reports the distribution of our submitted vulnerability reports across the tested projects. In total, we have submitted 54 reports, 53 were confirmed by developers. At the time of submission, 27 vulnerabilities have been fixed, with 37 new, unique CVE IDs assigned and 17 CVE applications pending.
5.1.2 Vulnerability types. Table 4 categorizes the vulnerabilities found by Concoction[3]. The top three security flaw-related categories are presented here. The "other types" category includes six types of vulnerabilities: 'allocation-size-too-big', 'out-of-memory', 'use-after-free', 'memcpy-param-overlap', 'illegal-memory-access',

[3]A full list can be found at https://github.com/HuantWang/CONCOCTION/blob/main/vul_info/README.md.

and 'DEADLYSIGNAL'. Of all the detected vulnerabilities, 62.3% are *buffer-overflow* related, covering both *heap* and *stack-buffer-overflow*. CONCOCTION's ability to detect buffer-overflow vulnerabilities comes from its capability to reason about input value change ranges by combining static code structures and carefully selected symbolic traces. During testing, CONCOCTION discovered six vulnerabilities (11.3%) related to *SEGV (segmentation violation)*, which were later confirmed by developers. CONCOCTION identifies *SEGV* by inferring and verifying bounds on the variable value of array references. Additionally, CONCOCTION submitted four vulnerabilities (7.5%) related to *memory-leaks*. The combination of static and dynamic code features enables CONCOCTION to infer this type of vulnerability by correlating the allocated and released memory buffer sizes within the test function. Listing 1 shows an example of a *heap-buffer-overflow* vulnerability detected by CONCOCTION, caused by an incomplete bounds-checking pattern. Meanwhile, Listing 2 presents a *SEGV* example uncovered by CONCOCTION.

5.1.3 *CONCOCTION detected examples.*

We present several examples of CONCOCTION-detected vulnerabilities, covering three types of memory-related security flaws. As DL models generally work as a black box [20], to understand CONCOCTION's workings and the vulnerability's root cause, we compare the original buggy code with the developer-generated patch after reporting the issue.

Heap buffer overflow. Listing 1 shows `CVE-2022-26181`, a *heap-buffer-overflow* vulnerability identified by CONCOCTION. The vulnerability stems from the value of the `data` variable. By making the `data` variable symbolic, CONCOCTION allows the DL model to infer that when `data` is not null, the execution trace consistently reaches line 6. CONCOCTION successfully identifies the incomplete bounds checking, which may overlook certain non-compliant inputs.

```
1     void aligned_dealloc(unsigned char *data) {
2       if (!data) return;
3       // function always_assert(condition) is used to catch
             exception when condition is ture
4   +   always_assert(((size_t)(data-0) & 0xf) == 0);
5   +   always_assert(data[-1] <= 0x10);
6       data -= data[-1];
7       custom_free(data);}
```

Listing 1: Patch for CVE-2022-26181, a `heap-buffer-overflow` vulnerability in the Lepton project.

Segmentation violation. Listing 2 shows a *segmentation-violation* vulnerability resulting from an out-of-bounds read. The issue arises when the value of `dctx->dcmpri.len` exceeds `dctx->dcmpri.f->len`, causing a reference to a memory location beyond the allocated buffer boundary. CONCOCTION could not discover this vulnerability without the symbolic trace input.

```
1     dctx->dcmpri.f->len = sizeof(dctx->dcmpri.f->rcache);
2     buf = dctx->dcmpri.f->rcache;
3   + if(dctx->dcmpri.len > dctx->dcmpri.f->len)
4   + { dctx->dcmpri.len = dctx->dcmpri.f->len; }
5   ... // pass variable dctx->dcmpri.len to buf_len
6     for(i=0; i<buf_len; i++) {...
7           b = buf[i];
8           ...}
```

Listing 2: Patch for a *segmentation-violation* vulnerability that reassigns `dctx->dcmpri.len` to avoid memory overflow.

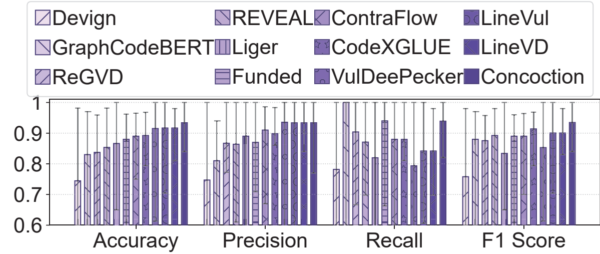


Figure 7: Evaluation on standard vulnerability databases. Min-max bars show performance across vulnerability types.

Memory leak. Listing 3 shows `CVE-2021-3574`, a *memory leak* vulnerability discovered by CONCOCTION. This issue can be detected by inspecting the execution trace from the memory allocation size of `samples_per_pixel` to the memory-free size of `MaxPixelChannels`. CONCOCTION detects this vulnerability by learning to compare the sizes of these two variables because `memory leaks` are typically caused by allocating more memory than required and subsequently freeing less.

```
1     malloc (samples_per_pixel) // malloc a buffer with size
             equal to variable samples_per_pixel
2     ...
3   + if (samples_per_pixel > MaxPixelChannels)  {
4   +     TIFFClose(tiff);
5   +     ThrowReaderException(CorruptImageError,
6   +     "MaximumChannelsExceeded");}
7     ...RelinquishMagickMemory(MaxPixelChannels) // free the
             buffer with size equal to variable MaxPixelChannels
```

Listing 3: Patch of CVE-2021-3574 for fixing a `memory leak` vulnerability in the ImageMagick project.

5.1.4 *Importance of bugs found.*

It is difficult to assess the importance of the bugs we found. Still, we found some evidence to show their importance: (1) some of the bugs we found were also reported by other application users later, indicating that the issues we identified are relevant and have occurred in real-world use cases; (2) most of our newly reported issues were confirmed and fixed by the developers demonstrates their importance; (3) developers promptly welcomed and resolved 14 of our reported issues within 48 hours, showing the importance of these issues.

5.2 Comparison on Open Datasets

5.2.1 *SARD dataset.*

Figure 7 reports four "higher-is-better" metrics (Sec. 4.3) achieved by CONCOCTION and the baselines on the SARD dataset (Table 1). The min-max bar shows variances across cross-validation runs. CONCOCTION outperforms other methods in all metrics and has the most reliable performance across cross-validation runs, with the narrowest min-max bar. While LINEVUL and LINEVD achieve high precision (low false-positive rate) similar to CONCOCTION, they have lower Recall and miss some vulnerable cases. For example, LINEVD only detects 70.3% of the CWE-126 vulnerability typed test cases, whereas CONCOCTION detects all. Other baselines show low detection accuracy. CONCOCTION achieved 100% recall in detecting certain vulnerability types like CWE-416 and CWE-789. Other methods, in contrast, failed to detect all of these vulnerabilities. Notably, CONCOCTION is highly effective in detecting the `use-after-free` vulnerability by leveraging dynamic traces and static code structures to infer the use of pointers.

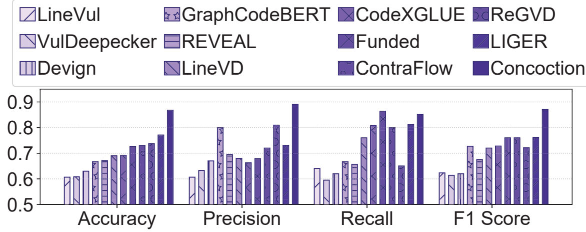


Figure 8: Evaluation on the CVE dataset.

Table 5: The number of vulnerabilities found by different methods for the projects in Table 2. The "#vuln." column shows the total vulnerabilities across all tools for a category.

Categories	Approaches	#vuln.
Static analysis tools	Infer [4], CodeQL [1]	5
Symbolic execution engines	KLEE [17], MoKLEE [16]	6
Fuzzing tool	AFL++ [31]	8
DL based on static code information	Vuldeepecker [57], Funded [80], Devign [88], ReVeal [21], ReGVD [63], LineVul [33], LineVD [43], CodeXGLUE [60], GraphCodeBERT [39], ContraFlow [23]	22
DL based on dynamic information	LIGER [81]	16
	Concoction	31

5.2.2 CVE dataset. As explained in Sec. 4.1, test samples in the CVE dataset (Table 1) are more complex than the SARD dataset. As such, it is more challenging to achieve good performance. However, Concoction outperforms all other methods across all evaluation metrics shown as Figure 8. Thanks to the carefully selected execution traces, Concoction can track changes in program states and variables (shown in examples in Sec. 5.1). This information enhances precision by reducing the false positive rate and helps discover more vulnerabilities with a higher true positive rate than static information alone, resulting in a higher recall. Among the baseline methods, LIGER performs best, but its F1 score is 10.4% lower than that of Concoction. This shows that Concoction strikes a better balance between false and negative positives, leveraging the advantages of structured static source code information.

5.3 Comparison on Known CVEs

We compare Concoction to the baselines on three open-source projects listed in Table 2. These projects contain 35 CVEs reported by independent users, which were also used by prior work [69, 71]. We apply all methods to functions associated with a CVE and use the reported CVEs to compute evaluation metrics (Sec. 4.3). To ensure a fair comparison, we train all methods, including Concoction, on the same training dataset, but we exclude these projects from the training data. For the dynamic methods listed in Sec. 4.2, we allocate 200 hours of machine time for each project.

5.3.1 Vulnerabilities identified. Table 5 demonstrates Concoction's advantages over other methods in identifying vulnerabilities across the three open-source projects evaluated. Concoction achieved 100% precision and 89% recall, correctly detecting 31 out of 35 confirmed CVEs. Additionally, Concoction identified all issues found by other methods and uncovered 9 additional vulnerabilities that others missed. Among DL-based static methods, ReGVD had the second-best and highest recall rates, detecting 21 vulnerabilities. However, they struggled to identify 14 vulnerabilities caused by API parameter misuse. Static tool baselines, CodeQL and Infer, can only detect five vulnerabilities at their best (5 for CodeQL and 0 for Infer) because they rely on hand-crafted rules with limited coverage. They missed all vulnerabilities related to "*CWE-754: Improper Check for Unusual or Exceptional Conditions*" because it was not in their rule sets. Dynamic methods using symbolic execution and fuzzing tools have a low recall of 0.26 in our evaluation due to limited path coverage within the testing time (12 hours). MoKLEE found 2 more bugs than KLEE (which found 4) in this context. Without Concoction's DL component, native AFL++ detected 8 vulnerabilities in 12 hours, while Concoction improved efficiency by finding 31 CVEs within the same test time by guiding the fuzzing engine to focus on potentially buggy code paths.

Though highly effective, Concoction missed four vulnerabilities (one example in Sec. 5.3.3) due to incomplete vulnerable execution traces during feature extraction. Addressing this limitation could involve extending symbolic execution time and improving the path selection model or the number of paths selected.

5.3.2 Case missed by baselines. Listing 4 shows `CVE-2020-35523`, an `integer overflow` vulnerability identified by Concoction but missed by all baselines. Other methods missed this bug because they mainly rely on static information, such as tokens typically associated with `integer overflow` (e.g., `malloc`), which is insufficient in this case. Concoction detects this vulnerability by symbolizing the `tw` and `w` variables, allowing the DL model to infer the absence of a corresponding `INT32` bounds check, which aligns with the pattern of `integer overflow`.

```
1     if (flip & FLIP_VERTICALLY) {
2  +    if ((tw + w) > INT_MAX) {
3  +      TIFFErrorExt(...);//
4  +      return (0);}
5       y = h - 1;
6       toskew = -(int32)(tw + w);
```

Listing 4: An `integer overflow` vulnerability in Libtiff.

5.3.3 Case missed by Concoction . List 5 shows `CVE-2020-35523`, a `memory leak` vulnerability missed by Concoction and all other baselines. The issue is caused by the function directly returning without closing the input file handle `in`, causing resource leakage. Concoction missed this because the memory leakage is introduced within the file handle data structure but Concoction does not learn such patterns from the training dataset. This can be improved by extending the training dataset to cover a wider range of patterns.

```
1     extern int optind;
2     in = TIFFOpen(argv[optind], "r");
3     ...
4     if (...) {
5       fprintf(...);
6  +    (void) TIFFClose(in);
7       return (-1);}
```

Listing 5: A `memory leak` vulnerability in Libtiff that occurs because the file handle `in` is not released.

5.4 Ablation Study

5.4.1 DL model implementation choices. We conduct an ablation study [36] on Concoction using the CVE dataset. The study includes the following variants: Static (using enhanced AST, Sec 3.2.1),

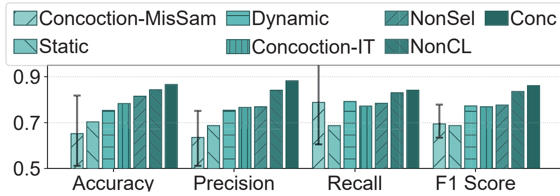


Figure 9: Concoction variants on CVE dataset.

Dynamic (utilizing randomly sampled symbolic traces with 30 minutes of symbolic execution for each project, Sec.3.2.2), NonCL (without the contrastive learning module, Sec.3.3), NonSel (omitting the path selection module by using randomly sampled symbolic execution traces with static code information, Sec 3.4), and Conc (the complete Concoction implementation).

The results are given in Figure 9. Using only static or dynamic representations is insufficient for accurately modeling program structures, with F1 scores of 68.7% and 77.2% for each variant, respectively. In our approach, we employed dropout-based contrastive learning as data augmentation for training our representation model (Sec.3.3.2). This helps extend our training set and mitigates overfitting [70]. Removing the contrastive learning component led to a 3.7% decrease in the F1 score, reaching 82.4% compared to the full model. Additionally, removing the path selection method resulted in an F1 score drop to 77.6% since random sampling may not capture crucial path information within a given budget.

5.4.2 Sensitivity analysis. To test the sensitivity of Concoction on mislabeled training samples, we introduce mislabeled samples that account for from 20% to 80% of the training samples into the training dataset of Concoction, leading to a variant of Concoction-MisSam. Similarly, we randomly remove some symbolic execution traces selected by Concoction to simulate a scenario where some traces are missing. This led to another implementation variant named Concoction-IT. The performance of Concoction-MisSam on Figure 9 shows that mislabeled training samples can harm performance, where the F1 score of Concoction-MisSam drops from 78.0% to 64.3%. This is expected, as machine learning techniques can suffer from noisy and mislabeled training data [30, 32]. However, the impact of mislabeled samples can be mitigated by increasing the training dataset and using data cleaning methods [13, 66], which are orthogonal to our approach. Similarly, missing execution traces can also negatively impact the performance, where the F1 score of Concoction decreases from 86.1% to 77.1% in Concoction-IT, which is still at least 2.1% higher than other DL baselines that rely on static code information. Missing symbolic execution traces is likely to happen when testing external libraries where the tool has no access to the source code. This issue is beyond the capability and scope of a source code-level detection tool. In the worst case, where all symbolic traces are missing, our DNN model can still use static information to detect bugs, albeit less efficiently.

5.4.3 Training and deployment overhead. Figure 10 compares training overhead for various DL-based methods. It includes one-off time spent on feature extraction (e.g., AST and symbolic executions) on training samples and iterative training time using labeled samples from Table 1. Training terminates when the loss does not improve within 20 consecutive epochs or meets the termination criteria specified in the baselines' source publication. The experiment was conducted on a multi-core server using a desktop-level NVIDIA 2080Ti GPU. Vuldeepecker, CodeXGLUE, and Lin *et al.* achieved the shortest training overhead, relying mainly on sequence neural networks like Bi-LSTM. However, they have a low F1 score, indicating a limited ability to capture complex code structures. More advanced models that use ASTs required longer feature extraction and training times but showed higher accuracy during evaluation. Additionally, LIGER and Concoction incur a more expensive feature extraction time due to collecting dynamic runtime information by symbolic execution. It is important to note that model training is performed offline and is a *one-off* cost.

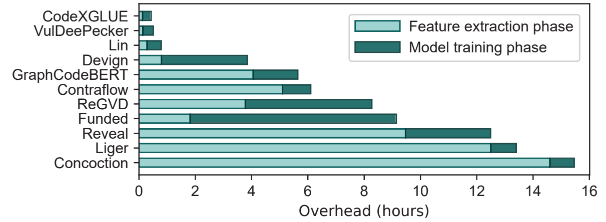


Figure 10: Training overhead of different DL methods.

During deployment, Concoction can complete predictions within minutes, and the fuzzing tool may take several hours to generate a vulnerability-exposing test case. Since Concoction can be integrated with a parallelized overnight build system, the deployment overhead should be acceptable for many software developers.

6 DISSCUSSIONS AND THREATS TO VALIDITY

Naturally, there is room for further work and improvement. We discuss a few points here.

Runtime overhead. The runtime overhead of Concoction mainly comes from collecting symbolic execution traces. The Concoction path selection component is designed to minimize this overhead. There are methods to accelerate symbolic executions through parallelizing symbolic test case generation [14, 28, 65]. These approaches are orthogonal to Concoction.

Large and complex code. During our evaluation of 23 large open-source projects, we used function inlining to remove the procedural boundaries for code analysis. Our current implementation does not support analysis for complex code involving recursive functions, pointer aliasing, and function pointers with dynamic dispatching, which remains an open problem. Concoction will benefit from techniques for reasoning these complex code patterns [24, 49, 52, 55, 68]. Our current implementation parallelizes symbolic executions and fuzzing on individual code regions. We envision that extending Concoction to larger code regions would require new techniques. For example, it would require capturing the data flow across methods and potentially pointer alias analysis [44], as well as techniques to accelerate symbolic executions [14, 28, 65]. We are excited about the potential of Concoction and hope its initial promising results can encourage further research.

Other languages. We showcased Concoction in C, but it can extend to other languages. Doing so requires adapting the source code rewriting tool (see Sec. 3.2.1) and having a tool to generate

symbolic traces for the target language. Nevertheless, our DL framework for combining static and dynamic information broadly applies to other languages. Concoction already supports using the Language Server Protocol (LSP) [38] to construct the enhanced AST. LSP currently supports C, C++, Java, JavaScript and Python with a single interface so Concoction can be easily ported to these languages. Concoction can also use symbolic execution engines built for other languages, including JDart [61] for Java, PyExZ3 for Python [45], and Jalangi2 [73] for JavaScript.

7 RELATED WORK

Our work builds on the past foundations of deep learning, source code vulnerability analysis, and static symbolic execution. We apply Transformer-based networks [39, 78] to learn program representations and contrastive learning to train our models.

Deep learning-based vulnerability detection. Our research is part of the recent efforts in DL-based software vulnerability detection [21, 57, 59, 60, 63]. Prior studies primarily relied on static information like ASTs and control flow graphs. Concoction incorporates symbolic traces with static code information to capture deeper program semantics, enhancing vulnerability detection.

Symbolic execution. Symbolic execution [17, 41] sidesteps the need for hand-crafted rules by exploiting symbolic values and analyzing their use over the execution tree of a program on source code. However, language constructs like loops and branches can significantly increase the number of execution states, limiting the scalability of the technique to large programs [11].

Contrastive learning. Contrastive learning is popular for its ability to reduce the costs of annotating large-scale datasets [46]. It has been used in computer vision [19, 22, 42], natural language processing [10, 35], and other domains. Concoction uses contrastive learning for addressing label scarcity in vulnerability detection. This allows us to train our models effectively with unsupervised learning, reducing the need for extensive manual labeling.

Path selection for code embedding. Several techniques use learning-based approaches for path selection [23, 41]. Like ContraFlow [23], Concoction also employ unsupervised active learning to train a path selection network. ContraFlow is designed to pinpoint static value-flow paths that may trigger a vulnerability, but it still solely relies on static code features. Unlike [23], we use path selection to minimize the overhead of symbolic executions, which are then combined with static code features to learn more efficient program representation. Nevertheless, our experiments in Sec. 5.2 and 5.3 show that our approach outperforms ContraFlow.

Machine learning in software engineering. Machine learning techniques have been powerful in software development [26, 27, 79, 82, 84]. Existing approaches address a variety of development tasks, including fuzz testing [31, 83], detecting code clone [29, 39, 48, 56], improving static analysis for vulnerability detection [55, 77], repairing programs [74], defect prediction [53, 85], attack detection [86] and processing vulnerability reports [25, 67, 87]. Concoction builds on those past foundations but is quite different from these studies.

8 CONCLUSION

We have presented Concoction, a new DL system for detecting vulnerabilities at the source code level. It utilizes structured static code features and dynamic symbolic execution traces to learn program representations, enabling accurate prediction of bugs. We train Concoction by combining unsupervised and supervised learning and minimizing the overhead of symbolic executions by using a path selection network. We apply Concoction to detect bugs and vulnerabilities for C programs from 20 open-source projects. In 200 hours of automated concurrent test runs, Concoction successfully detected vulnerabilities in all tested projects, discovering 54 unique vulnerabilities and yielding 37 new, unique CVE IDs. Compared to 16 previous methods, Concoction finds more vulnerabilities with higher accuracy and a lower false positive rate.

ACKNOWLEDGMENTS

This work was supported in part by the National Natural Science Foundation of China (NSFC) under grant agreements 61972314 and 62372373, the Shaanxi International Science and Technology Cooperation Program (2023-GHZD-04), the Shaanxi Province "Engineers + Scientists" Team Building Program (2023KXJ-055), the Mitacs Globalink Research Award (GRA) program and the UK Engineering and Physical Sciences Research Council (EPSRC) under grant agreement EP/X018202/1.

For the purpose of open access, the author has applied a Creative Commons Attribution (CCBY) license to any Author Accepted Manuscript version arising from this submission. For any correspondence, please contact Zhanyong Tang (Email: zytang@nwu.edu.cn) and Zheng Wang (Email: z.wang5@leeds.ac.uk).

REFERENCES

[1] [n. d.]. CodeQL, discover vulnerabilities with semantic code analysis engine. https://codeql.github.com/
[2] [n. d.]. Common Vulnerabilities and Exposures (CVE). https://cve.mitre.org/.
[3] [n. d.]. Common Weakness Enumeration. https://cwe.mitre.org/.
[4] [n. d.]. Infer, a static program analyzer. https://fbinfer.com/docs/about-Infer
[5] [n. d.]. Jenkins, open source automation server. https://www.jenkins.io/
[6] [n. d.]. Joern(Open-Source Code Querying Engine for C/C++.). https://joern.io/.
[7] [n. d.]. OSS-Fuzz. https://github.com/google/oss-fuzz.
[8] Elena N Akimova, Alexander Yu Bersenev, Artem A Deikov, Konstantin S Kobylkin, Anton V Konygin, Ilya P Mezentsev, and Vladimir E Misilov. 2021. A survey on software defect prediction using deep learning. *Mathematics* 9, 11 (2021), 1180.
[9] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2018. Learning to Represent Programs with Graphs. In *Proceedings of the ICLR*.
[10] Sanjeev Arora, Hrishikesh Khandeparkar, Mikhail Khodak, Orestis Plevrakis, and Nikunj Saunshi. 2019. A theoretical analysis of contrastive unsupervised representation learning. *arXiv preprint arXiv:1902.09229* (2019).
[11] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. 2018. A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)* 51, 3 (2018), 1–39.
[12] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2329–2344.
[13] Carla E Brodley and Mark A Friedl. 1999. Identifying mislabeled training data. *Journal of artificial intelligence research* 11 (1999), 131–167.
[14] Stefan Bucur, Vlad Ureche, Cristian Zamfir, and George Candea. 2011. Parallel Symbolic Execution for Automated Real-World Software Testing. In *Proceedings of the Sixth Conference on Computer Systems* (Salzburg, Austria) *(EuroSys '11)*. Association for Computing Machinery, New York, NY, USA, 183–198. https://doi.org/10.1145/1966445.1966463
[15] Frank Busse, Pritam Gharat, Cristian Cadar, and Alastair F Donaldson. 2022. Combining static analysis error traces with dynamic symbolic execution (experience paper). In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. 568–579.

[16] Frank Busse, Martin Nowack, and Cristian Cadar. 2020. Running symbolic execution forever. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 63–74.

[17] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. 2008. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs.. In *OSDI*, Vol. 8. 209–224.

[18] Cristian Cadar and Koushik Sen. 2013. Symbolic execution for software testing: three decades later. *Commun. ACM* 56, 2 (2013), 82–90.

[19] Mathilde Caron, Ishan Misra, Julien Mairal, Priya Goyal, Piotr Bojanowski, and Armand Joulin. 2020. Unsupervised Learning of Visual Features by Contrasting Cluster Assignments. https://doi.org/10.48550/ARXIV.2006.09882

[20] Davide Castelvecchi. 2016. Can we open the black box of AI? *Nature News* 538, 7623 (2016), 20.

[21] Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. 2021. Deep learning based vulnerability detection: Are we there yet. *IEEE Transactions on Software Engineering* (2021).

[22] Ting Chen, Simon Kornblith, Mohammad Norouzi, and Geoffrey Hinton. 2020. A simple framework for contrastive learning of visual representations. In *International conference on machine learning*. PMLR, 1597–1607.

[23] Xiao Cheng, Guanqin Zhang, Haoyu Wang, and Yulei Sui. 2022. Path-Sensitive Code Embedding via Contrastive Learning for Software Vulnerability Detection. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, South Korea) *(ISSTA 2022)*. Association for Computing Machinery, New York, NY, USA, 519–531. https://doi.org/10.1145/3533767.3534371

[24] Khushboo Chitre, Piyus Kedia, and Rahul Purandare. 2022. The road not taken: exploring alias analysis based optimizations missed by the compiler. *Proceedings of the ACM on Programming Languages* 6, OOPSLA2 (2022), 786–810.

[25] Nathan Cooper, Carlos Bernal-Cárdenas, Oscar Chaparro, Kevin Moran, and Denys Poshyvanyk. 2021. It takes two to tango: Combining visual and textual information for detecting duplicate video-based bug reports. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 957–969.

[26] Chris Cummins, Pavlos Petoumenos, Zheng Wang, and Hugh Leather. 2017. End-to-end deep learning of optimization heuristics. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 219–232.

[27] Chris Cummins, Pavlos Petoumenos, Zheng Wang, and Hugh Leather. 2017. Synthesizing benchmarks for predictive modeling. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 86–99.

[28] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Budapest, Hungary) *(TACAS'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 337–340.

[29] Chunrong Fang, Zixi Liu, Yangyang Shi, Jeff Huang, and Qingkai Shi. 2020. Functional code clone detection with syntax and semantics fusion learning. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 516–527.

[30] Farzaneh S. Fard, Paul Hollensen, Stuart Mcilory, and Thomas Trappenberg. 2017. Impact of biased mislabeling on learning with deep networks. In *2017 International Joint Conference on Neural Networks (IJCNN)*. 2652–2657. https://doi.org/10.1109/IJCNN.2017.7966180

[31] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++: Combining Incremental Steps of Fuzzing Research. In *Proceedings of the 14th USENIX Conference on Offensive Technologies (WOOT'20)*. USENIX Association, USA, Article 10, 1 pages.

[32] Benoît Frénay and Michel Verleysen. 2013. Classification in the presence of label noise: a survey. *IEEE transactions on neural networks and learning systems* 25, 5 (2013), 845–869.

[33] Michael Fu and Chakkrit Tantithamthavorn. 2022. Linevul: A transformer-based line-level vulnerability prediction. In *Proceedings of the 19th International Conference on Mining Software Repositories*. 608–620.

[34] Tianyu Gao, Xingcheng Yao, and Danqi Chen. 2021. SimCSE: Simple Contrastive Learning of Sentence Embeddings. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Online and Punta Cana, Dominican Republic, 6894–6910. https://doi.org/10.18653/v1/2021.emnlp-main.552

[35] John Giorgi, Osvald Nitski, Bo Wang, and Gary Bader. 2020. Declutr: Deep contrastive learning for unsupervised textual representations. *arXiv preprint arXiv:2006.03659* (2020).

[36] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. 2014. Rich feature hierarchies for accurate object detection and semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 580–587.

[37] Weiwei Gu, Aditya Tandon, Yong-Yeol Ahn, and Filippo Radicchi. 2021. Principled approach to the selection of the embedding dimension of networks. *Nature Communications* 12, 1 (2021), 3772.

[38] Nadeeshaan Gunasinghe and Nipuna Marcus. 2021. *Language Server Protocol and Implementation.* Springer.

[39] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin B. Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2021. GraphCodeBERT: Pre-training Code Representations with Data Flow. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net. https://openreview.net/forum?id=jLoC4ez43PZ

[40] Hazim Hanif, Mohd Hairul Nizam Md Nasir, Mohd Faizal Ab Razak, Ahmad Firdaus, and Nor Badrul Anuar. 2021. The rise of software vulnerability: Taxonomy of software vulnerabilities detection and machine learning approaches. *Journal of Network and Computer Applications* 179 (2021), 103009.

[41] Jingxuan He, Gishor Sivanrupan, Petar Tsankov, and Martin Vechev. 2021. Learning to Explore Paths for Symbolic Execution. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 2526–2540.

[42] Kaiming He, Haoqi Fan, Yuxin Wu, Saining Xie, and Ross Girshick. 2020. Momentum contrast for unsupervised visual representation learning. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 9729–9738.

[43] David Hin, Andrey Kan, Huaming Chen, and M Ali Babar. 2022. LineVD: Statement-level vulnerability detection using graph neural networks. In *Proceedings of the 19th International Conference on Mining Software Repositories*. 596–607.

[44] Michael Hind, Michael Burke, Paul Carini, and Jong-Deok Choi. 1999. Interprocedural pointer alias analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 21, 4 (1999), 848–894.

[45] M Irlbeck et al. 2015. Deconstructing dynamic symbolic execution. *Dependable Software Systems Engineering* 40, 2015 (2015), 26.

[46] Ashish Jaiswal, Ashwin Ramesh Babu, Mohammad Zaki Zadeh, Debapriya Banerjee, and Fillia Makedon. 2020. A survey on contrastive self-supervised learning. *Technologies* 9, 1 (2020), 2.

[47] Prannay Khosla, Piotr Teterwak, Chen Wang, Aaron Sarna, Yonglong Tian, Phillip Isola, Aaron Maschinot, Ce Liu, and Dilip Krishnan. 2020. Supervised contrastive learning. *Advances in Neural Information Processing Systems* 33 (2020), 18661–18673.

[48] Seulbae Kim, Seunghoon Woo, Heejo Lee, and Hakjoo Oh. 2017. Vuddy: A scalable approach for vulnerable code clone discovery. In *Proceedings of the 2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 595–614.

[49] Sun Hyoung Kim, Dongrui Zeng, Cong Sun, and Gang Tan. 2022. Binpointer: towards precise, sound, and scalable binary-level pointer analysis. In *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction*. 169–180.

[50] Chris Lattner. 2008. LLVM and Clang: Next generation compiler technology. In *The BSD conference*, Vol. 5. 1–20.

[51] Changsheng Li, Handong Ma, Zhao Kang, Ye Yuan, Xiao-Yu Zhang, and Guoren Wang. 2021. On deep unsupervised active learning. In *Proceedings of the Twenty-Ninth International Conference on International Joint Conferences on Artificial Intelligence*. 2626–2632.

[52] Guoren Li, Hang Zhang, Jinmeng Zhou, Wenbo Shen, Yulei Sui, and Zhiyun Qian. 2023. A hybrid alias analysis and its application to global variable protection in the linux kernel. In *32nd USENIX Security Symposium (USENIX Security 23)*. 4211–4228.

[53] Ning Li, Martin Shepperd, and Yuchen Guo. 2020. A systematic review of unsupervised learning techniques for software defect prediction. *Information and Software Technology* 122 (2020), 106287.

[54] Yi Li, Shaohua Wang, Tien N Nguyen, and Son Van Nguyen. 2019. Improving bug detection via context-based code representation learning and attention-based neural networks. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–30.

[55] Zhuohua Li, Jincheng Wang, Mingshen Sun, and John CS Lui. 2021. MirChecker: detecting bugs in Rust programs via static analysis. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 2183–2196.

[56] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Hanchao Qi, and Jie Hu. 2016. VulPecker: an automated vulnerability detection system based on code similarity analysis. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*. 201–213.

[57] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. 2018. VulDeePecker: A Deep Learning-Based System for Vulnerability Detection. *Proceedings of the NDSS* (2018).

[58] Guanjun Lin, Sheng Wen, Qing-Long Han, Jun Zhang, and Yang Xiang. 2020. Software vulnerability detection using deep neural networks: a survey. *Proc. IEEE* 108, 10 (2020), 1825–1848.

[59] Guanjun Lin, Jun Zhang, Wei Luo, Lei Pan, Olivier De Vel, Paul Montague, and Yang Xiang. 2019. Software vulnerability discovery via learning multi-domain knowledge bases. *Proceedings of the IEEE Transactions on Dependable and Secure Computing* (2019).

[60] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. 2021. CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. In *Thirty-fifth Conference on Neural Information Processing*

Systems Datasets and Benchmarks Track.

[61] Kasper Luckow, Marko Dimjašević, Dimitra Giannakopoulou, Falk Howar, Malte Isberner, Temesghen Kahsai, Zvonimir Rakamarić, and Vishwanath Raman. 2016. JD art: a dynamic symbolic analysis framework. In *Tools and Algorithms for the Construction and Analysis of Systems: 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings 22.* Springer, 442–459.

[62] Andrew A Neath and Joseph E Cavanaugh. 2012. The Bayesian information criterion: background, derivation, and applications. *Wiley Interdisciplinary Reviews: Computational Statistics* 4, 2 (2012), 199–203.

[63] Van-Anh Nguyen, Dai Quoc Nguyen, Van Nguyen, Trung Le, Quan Hung Tran, and Dinh Phung. 2022. ReGVD: Revisiting Graph Neural Networks for Vulnerability Detection. In *Proceedings of the 44th International Conference on Software Engineering Companion (ICSE '22 Companion).*

[64] NIST. [n. d.]. Software Assurance Reference Dataset Project. https://samate.nist.gov/SRD/.

[65] Hristina Palikareva and Cristian Cadar. 2013. Multi-solver support in symbolic execution. In *Computer Aided Verification: 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings 25.* Springer, 53–68.

[66] Xiaojiang Peng, Kai Wang, Zhaoyang Zeng, Qing Li, Jianfei Yang, and Yu Qiao. 2020. Suppressing mislabeled data via grouping and self-attention. In *Computer Vision–ECCV 2020: 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part XVI 16.* Springer, 786–802.

[67] Henning Perl, Sergej Dechand, Matthew Smith, Daniel Arp, Fabian Yamaguchi, Konrad Rieck, Sascha Fahl, and Yasemin Acar. 2015. Vccfinder: Finding potential vulnerabilities in open-source projects to assist code audits. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security.* 426–437.

[68] Gabriel Poesia and Fernando Magno Quintão Pereira. 2020. Dynamic dispatch of context-sensitive optimizations. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–28.

[69] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. 2015. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis.* 24–36.

[70] Leslie Rice, Eric Wong, and Zico Kolter. 2020. Overfitting in adversarially robust deep learning. In *Proceedings of the 37th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 119)*, Hal Daumé III and Aarti Singh (Eds.). PMLR, 8093–8104. https://proceedings.mlr.press/v119/rice20a.html

[71] Manuel Rigger and Zhendong Su. 2020. Testing Database Engines via Pivoted Query Synthesis.. In *OSDI*, Vol. 20. 667–682.

[72] Iqbal H Sarker. 2021. Machine learning: Algorithms, real-world applications and research directions. *SN Computer Science* 2, 3 (2021), 1–21.

[73] Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. 2013. Jalangi: A Selective Record-Replay and Dynamic Analysis Framework for JavaScript. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering* (Saint Petersburg, Russia) *(ESEC/FSE 2013).* Association for Computing Machinery, New York, NY, USA, 488–498. https://doi.org/10.1145/2491411.2491447

[74] Ridwan Shariffdeen, Yannic Noller, Lars Grunske, and Abhik Roychoudhury. 2021. Concolic program repair. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation.* 390–405.

[75] Ke Shi, Yang Lu, Jingfei Chang, and Zhen Wei. 2020. PathPair2Vec: An AST path pair-based code representation method for defect prediction. *Journal of Computer Languages* 59 (2020), 100979. https://doi.org/10.1016/j.cola.2020.100979

[76] Wilson L Taylor. 1953. "Cloze procedure": A new tool for measuring readability. *Journalism quarterly* 30, 4 (1953), 415–433.

[77] David A Tomassi and Cindy Rubio-González. 2021. On the Real-World Effectiveness of Static Bug Detectors at Finding Null Pointer Exceptions. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE).* IEEE, 292–303.

[78] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

[79] Huanting Wang, Zhanyong Tang, Cheng Zhang, Jiaqi Zhao, Chris Cummins, Hugh Leather, and Zheng Wang. 2022. Automating reinforcement learning architecture design for code optimization. In *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction.* 129–143.

[80] H. Wang, G. Ye, Z. Tang, S. H. Tan, S. Huang, D. Fang, Y. Feng, L. Bian, and Z. Wang. 2021. Combining Graph-Based Learning With Automated Data Collection for Code Vulnerability Detection. *IEEE Transactions on Information Forensics and Security* 16 (2021), 1943–1958. https://doi.org/10.1109/TIFS.2020.3044773

[81] Ke Wang and Zhendong Su. 2020. Blended, Precise Semantic Program Embeddings. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020).* Association for Computing Machinery, New York, NY, USA, 121–134. https://doi.org/10.1145/3385412.3385999

[82] Zheng Wang and Michael O'Boyle. 2018. Machine learning in compiler optimization. *Proc. IEEE* 106, 11 (2018), 1879–1901.

[83] Guixin Ye, Zhanyong Tang, Shin Hwei Tan, Songfang Huang, Dingyi Fang, Xiaoyang Sun, Lizhong Bian, Haibo Wang, and Zheng Wang. 2021. Automated conformance testing for JavaScript engines via deep compiler fuzzing. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation.* 435–450.

[84] Guixin Ye, Zhanyong Tang, Huanting Wang, Dingyi Fang, Jianbin Fang, Songfang Huang, and Zheng Wang. 2020. Deep program structure modeling through multi-relational graph-based learning. In *Proceedings of the ACM International conference on parallel architectures and compilation techniques.* 111–123.

[85] Zhengran Zeng, Yuqun Zhang, Haotian Zhang, and Lingming Zhang. 2021. Deep just-in-time defect prediction: how far are we?. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis.* 427–438.

[86] Dan Zhang, Qing-Guo Wang, Gang Feng, Yang Shi, and Athanasios V Vasilakos. 2021. A survey on attack detection, estimation and control of industrial cyber–physical systems. *ISA transactions* 116 (2021), 1–16.

[87] Yu Zhao, Tingting Yu, Ting Su, Yang Liu, Wei Zheng, Jingzhi Zhang, and William GJ Halfond. 2019. Recdroid: automatically reproducing android application crashes from bug reports. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE).* IEEE, 128–139.

[88] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In *Proceedings of the Advances in Neural Information Processing Systems.* 10197–10207.