HYDRA: A Hybrid Heuristic-Guided Deep Representation Architecture for Predicting Latent Zero-Day Vulnerabilities in Patched Functions

Mohammad Farhad, Sabbir Rahman, Shuvalaxmi Dass
University of Louisiana at Lafayette
Lafayette, Louisiana, USA
{mohammad.farhad1,sabbir.rahman1,shuvalaxmi.dass}@louisiana.edu

Abstract

Software security testing, particularly when enhanced with deep learning models, has become a powerful approach for improving software quality, enabling faster detection of known flaws in source code. However, many approaches miss post-fix latent vulnerabilities that remain even after patches typically due to incomplete fixes or overlooked issues may later lead to zero-day exploits. In this paper, we propose **HYDRA**, a ***Hy**brid heuristic-guided **D**eep **R**epresentation **A**rchitecture* for predicting latent zero-day vulnerabilities in patched functions that combines rule-based heuristics with deep representation learning to detect latent risky code patterns that may persist after patches. It integrates static vulnerability rules, GraphCodeBERT embeddings, and a Variational Autoencoder (VAE) to uncover anomalies often missed by symbolic or neural models alone. We evaluate HYDRA in an unsupervised setting on patched functions from three diverse real-world software projects: Chrome, Android, and ImageMagick. Our results show HYDRA **predicts** 13.7%, 20.6%, and 24% of functions from Chrome, Android, and ImageMagick respectively as containing latent risks, including both heuristic matches and cases without heuristic matches (None) that may lead to zero-day vulnerabilities. It outperforms baseline models that rely solely on regex-derived features or their combination with embeddings, uncovering truly risky code variants that largely align with known heuristic patterns. These results demonstrate HYDRA's capability to surface hidden, previously undetected risks, advancing software security validation and supporting proactive zero-day vulnerabilities discovery.

Keywords

Zero-Day, Code Analysis, Patched Function, Deep Representation Learning, GraphCodeBERT, Vulnerability Prediction, Software Security.

1 INTRODUCTION

Software vulnerabilities have emerged as a constant and dynamic threat as a result of the complexity and inter-connectivity of modern software system [25, 45]. In response, software security testing has long been a cornerstone of software quality assurance, providing systematic and repeatable methods to verify software correctness, robustness, and reliability. As software systems continue to evolve rapidly, security testing has become essential for ensuring resilience and reliability throughout the software development lifecycle [18, 35, 38]. Despite these advancements in security testing and validation, zero-day vulnerabilities continue to pose a biggest risks to software security that are unpatched at the time of discovery and unknown to vendor. More formally, **Zero-day vulnerabilities** refer to previously unknown security flaws in software that are exploited before developers have had an opportunity to identify, patch, or disclose them [7]. These vulnerabilities often arise due to overlooked edge cases, incomplete patches, or misunderstandings of complex system behavior. As no official fix exists at the time of exploitation, zero-day attacks pose significant risks particularly in critical infrastructure, IoT, and open-source ecosystems. High profile incidents such as Stuxnet in 2010 [4] and SolarWinds Orion in 2020 [3] underscore the destructive potential of zero-day exploits and the importance of proactive vulnerability auditing. They enable hackers to take advantage of vulnerabilities before remedy is available, giving rise to silent, targeted attacks with devastating effects on users and systems [34, 43]. Open-source companies sometimes decide to patch vulnerabilities covertly without revealing them through formal advisories like CVE [11] reports because of worries about maintainability and reputation. By converting zero-day risks into so called n-day vulnerabilities, this silent patching technique allows adversaries to target unpatched installations and reverse-engineer patches before downstream consumers can apply updates [40]. However most traditional methods like static analysis tools (e.g., Fortify) and signature based scanners (e.g. CodeQL) are built to identify known patterns of insecure code, despite the fact that researchers have made impressive strides in creating deep learning based vulnerability detectors.

Using a variety of techniques, researchers have tried to find these hidden defects. For instance anomaly detection and autoencoder based techniques have been investigated to capture latent behavioral patterns of zero-day threats in network traffic [27], while machine learning classifiers have been trained to differentiate between benign code changes and secret security patches [40]. To describe patch timescales and programming language risks, others have been conducted empirical analysis of open source software repositories [43]. However, the majority of these methods either use post-disclosure datasets or target network traffic or binary code, losing the proactive chance to reason about already fixed functions. In order to identify potential residual vulnerabilities, recent studies have emphasized the importance and practicality of mining silent patches (i.e., fixes that are applied without public disclosure) and tracing them back to their root causes or original vulnerable code.

For instance, GraphSPD [39] and GRAPE [16] use code property graphs and Graph Convulational Networks to detect undocumented vulnerability patches by analyzing structural differences. While effective, they frequently make the assumption that both pre-patch and post-patch code are available. DeepDFA [36] extends vulnerability semantics using control/data-flow graphs and deep

First published: arXiv:2511.06220v1 9 Nov 2025

learning, achieving strong generalization and fast training, but still demands substantial structural modeling. Generative models like VulRepair [14] and Zero-Shot Repair [31] utilize large pre-trained language models (e.g., T5, Codex) to rewrite buggy code into secure versions. However, these methods focus on repair rather than detection and often depend on test suites, human oversight, or complex prompts. In contrast, data centric approaches such as VULGEN [30] and VGX [29] generate synthetic vulnerable code via contextual transformations, while UL-VAE [6] applies unsupervised anomaly detection to identify zero-day patterns in IoT malware. CLNX [32] improves CodeBERT's robustness by incorporating commit context, and augmentation strategies help reduce overfitting in models like GraphCodeBERT.

Despite these advances, most existing methods operate on vulnerable code snapshots, commit diffs, or symbolic traces of unpatched code. In contrast, our work explores whether patched functions assumed secure may still contain latent indicators of vulnerabilities, posing risks of re-emergent or zero-day exploits. In this work, we pose a central question: *Can latent vulnerability patterns persist in patched functions previously considered secure thereby introducing future zero-day risks?* To investigate this, we introduce **HYDRA**, a hybrid **prediction** framework to improve post-patch software security testing, that combines domain-driven heuristic rules with semantic embeddings from a pre-trained model.

Motivated by prior findings on hidden or misclassified patches in open-source software [40] and the significance of silent fixes in long term code maintenance [43], we hypothesize that many patches may leave behind residual vulnerabilities either due to incomplete remediation or rushed developer fixes. HYDRA is designed to uncover these overlooked risks by integrating two complementary components: **(a). Symbolic pattern matching**, based on five manually defined heuristic rules implemented using regular expressions (regex), and **(b). Semantic embeddings** generated from GraphCodeBERT [15], which captures both token level context and data flow graph (DFG) information. While regex offers interpretability and efficiency long used in static analysis to flag insecure code, as seen in VGX [29] it is inherently limited to predefined patterns. In contrast, GraphCodeBERT enables HYDRA to generalize beyond symbolic rules, learning deeper semantic cues that may indicate risky behavior even when explicit patterns are absent. Notably, our model can label functions as `None` when no heuristic rule matches, highlighting previously patched function segments that may contain unknown or emergent zero-day risks. By combining symbolic reasoning with deep contextual understanding, HYDRA achieves both transparency in **prediction** and the generalization capacity of modern language models offering a novel perspective on post-patch vulnerability auditing.

In summary, this paper makes the following **key contributions**:

- We highlight the challenge of predicting latent zero-day vulnerabilities through common heuristic patterns that persist in previously patched functions a critical blind spot often overlooked by both static analysis and deep learning approaches.(**Section** 2)
- We propose HYDRA, a novel hybrid unsupervised architecture that integrates symbolic heuristic rules with deep GraphCodeBERT embeddings and a Variational Autoencoder (VAE), enabling effective **prediction** and clustering of residual risky code patterns without reliance on external program artifacts like ASTs or diff logs.(**Section** 3)
- We evaluate HYDRA on patched functions from Chrome, Android, and ImageMagick, showing its ability to detect latent vulnerabilities achieving up to 4× higher Silhouette, 10.1× higher CHI, and 72.2% lower DBI than baselines while flagging 13–24% of functions as potentially vulnerable, including those without explicit rule matches.(**Section** 4)
- We further demonstrate HYDRA's practical utility by identifying risky patterns through two real-world case studies from the Chrome and ImageMagick projects.(**Section** 5)

Sections 6, 7 and 8 mention about the threats to validity, related work and conclusion along with future work respectively.

2 BACKGROUND & MOTIVATION

This section presents background and motivating examples that highlight the limitations of traditional tools such as static analyzers and CVE trained classifiers (see related work 7) in predicting residual vulnerabilities in source code. These examples represent the risky coding patterns that expose gaps in identifying post-patch risks, which HYDRA later revisits to demonstrate its ability to uncover, paving the way for zero-day vulnerability **prediction**.

2.1 Background

Since downstream users might not be aware of unresolved or partially repaired vulnerabilities, latent vulnerability patches those done without disclosure, pose serious risks [37]. This motivates post-fix analysis frameworks that can verify the integrity of already fixed patches. Reducing the attack surface for unknown or new threats requires identifying code that has been patched but still has latent weaknesses. Even though structural embeddings can help identify unusual behaviour [24], however, there is currently no method, to the best of our knowledge, for identifying latent risk vectors in patched functions that combines learned semantic embeddings with domain heuristics.

In manual code review and static analysis, rule based heuristics such as race conditions, unsafe memory allocation, and missing null checks etc have long been used to find recurring patterns in a code [26, 42]. Each of these heuristics is associated to well known Common Weakness Enumerations (CWEs) IDs [10], a list of common software and hardware weaknesses that can lead to vulnerabilities such as CWE-476 (NULL Pointer Deference) or CWE-119 (Buffer Overflow). These heuristics also frequently appear in Common Vulnerabilities and Exposures (CVEs) [11], a system that provides a standardized identifier for publicly known vulnerabilities, facilitating traceability and comparison across projects indicating their critical relevance in real-world security patches. In order to improve code semantic representation, GraphCodeBERT [15] is a cutting edge pretrained transformer model that combines source code with data flow graph (DFG) structures. It allows structural reasoning without explicit CFG, AST inputs by internally constructing and encoding DFGs, even when we simply input source code. HYDRA introduces hybrid design that merges handcrafted vulnerability patterns with the representational strength of pre-trained models

like GraphCodeBERT.

2.2 Motivating Examples and Challenges

HYDRA is motivated by the frequent occurrence of residual risky heuristic patterns in post-patch functions. We present **five** main heuristic rules derived from most common insecure coding patterns observed in real-world OSS and CVE linked patches [1]. While HYDRA currently employs these five heuristics, its module is designed to be extensible, allowing incorporation of additional rules for broader generalization. In our work, we chose these five motivating examples as these rules capture flaws most often introduced by human error, poor fix localization, or misinterpretation of vulnerability contexts [9], hence they reveal key challenges associated with residual risky patterns in patched functions. They motivate our hybrid approach by revealing persistent risk patterns even in patched function. Though developed independently from empirical observations, these rules later aligned with major Linux Kernel vulnerability classes [9], such as buffer overflows and null pointer dereferences underscoring their practical relevance and generality. The risky heuristic patterns and associated challenges are defined below:

Missing Null Check: In Figure 1a, the `rtt_reset` function in line 1 accepts a pointer *sk* and immediately passes it into transformations functions `tcp_sk(sk)` and `inet_csk_ca(sk)` (see line 2 and 3). The result of those functions (tp, ca) are used without any nullity verification. This dereferenced pointers without null checks, which is a common bug type in real-world vulnerability datasets (e,g, CWE-476). This leads to *information disclosure* and *denial of service (DoS)*, where dereferencing unchecked pointers can allow unintended access to sensitive memory regions or privileged operations.

Unsafe Memory Allocation: The use of `calloc` (line 3) in Figure 1b, without checking its return value introduces risk of dereferencing a null pointer if memory allocation fails. If `calloc` returns `NULL` and the subsequent `strcat` (line 4 and 5 highlighted) operation is invoked on this null pointer, it will lead to undefined behavior, likely causing segmentation faults or process crashes. Such patterns align with *CWE-690: Unchecked Return Value to NULL Pointer Dereference*, which emphasizes the importance of validating the success of memory allocation before using the pointer. It leads to *memory corruption and manipulations*, where unchecked allocation results can cause undefined behavior, data overwrites or exploitation through crafted inputs.

Logging Without Halting: The function `fprintf(stderr, ...)` (line 4) attempts to log an error condition when a configuration file fails to open in Figure 1c. However, it does not perform any blocking or termination behavior afterward such as returning an error code or exiting the program thus allowing execution to proceed in an invalid or undefined state. This may result in dereferencing `client->response_code` which may be uninitialized or invalid, depending on the `http_send` outcome. This issue corresponds to *CWE-703: Improper Check or Handling of Exceptional Conditions*, where an error is acknowledged but not adequately acted upon, leading to continued use of an invalid program state. Due to this, might happen *privilege escalation*, as the program continues execution in invalid states after logging errors, potentially leaking sensitive data or triggering crashes.

Missing Bounds Check: In Figure 1d, line 4 (highlighted) accesses `kvp_file_info[pool]`, referencing an array or struct pointer. However, the function lacks validation to ensure that pool is within bounds (e.g.,$0 \leq$ `pool` $<$ `MAX_POOLS`) or that `kvp_file_info[pool].fd` is a valid file descriptor (e.g.,≥ 0). If pool is not properly validated before invocation, it may lead to undefined behavior or memory corruption. Moreover, using an invalid `fd` in `fcntl` (line 5) could result in runtime errors, particularly if the value is uninitialized posing a risk consistent with CWE-125 (Out-of-Bounds Read). This conducts to *buffer overflows* and *injection attacks*, enabling attackers to overwrite memory or inject malicious payloads via invalidated array or pointer access.

Race Condition: In Figure 1e, the `update_user_profile` function (line 1) modifies a shared user object without any synchronization. In multi-threaded environments, concurrent invocations on the same user instance can lead to unsafe interleaving e.g., during the `strcpy` operation (line 3) or while updating `last_updated`. This may result in inconsistent logs, corrupted state, or memory safety issues depending on subsequent use. The lack of synchronization primitives (e.g., `pthread_mutex_lock`) constitutes a classic data race, aligning with CWE-362: Concurrent Execution using Shared Resource with Improper Synchronization. It may operates to *deadlocks* and *inconsistent system states*, as lack of synchronization in multi-threaded environments may corrupt shared data or halt program progress.

3 METHODOLOGY

This section outlines the proposed approach of HYDRA. After giving a general introduction to HYDRA, we go into details of each module.

3.1 Overview of HYDRA

Identifying latent vulnerabilities especially those that persist after patching is a critical yet underexplored challenge in software security [16, 34, 37, 39]. While many detection systems leverage deep neural models to learn from source code [24, 28, 39, 44], few tackle the nuanced task of predicting future zero-day risks in patched code, where rushed fixes or incomplete understanding may leave exploitable remnants. To address this, we propose **HYDRA** (***Hy**brid heuristic-guided **D**eep **R**epresentation **A**rchitecture*), a novel hybrid framework that bridges deep semantic reasoning with symbolic pattern explainability. HYDRA integrates *regex based heuristic prediction* with *GraphCodeBERT* [15] to capture both structural and contextual code semantics. An unsupervised clustering module (e.g., K-Means, VAE) further enhances **prediction** by surfacing outlier patterns potential indicators of unknown or non obvious risks beyond predefined rules. We will now describe the HYDRA architetcure in detail.

3.2 The HYDRA Architecture

HYDRA is a hybrid vulnerability **prediction** framework designed to identify latent zero-day risk indicators in already patched functions. As shown in Figure 2, it has two phases: Learning and Testing.

I. Learning Phase. In this phase, HYDRA ingests post-fix functions which are passed through two parallel process pipelines: one *heuristic-driven* and the other based on *semantic embeddings*. The Learning pipeline consists of the following steps:

```c
static void rtt_reset(struct sock *sk) {
 struct tcp_sock *tp = tcp_sk(sk);
 struct illinois *ca = inet_csk_ca(sk);
 ca->end_seq = tp->snd_nxt;
 ca->cnt_rtt = 0;
 ca->sum_rtt = 0;}
```

(a) Missing Null Check

```c
char* prepare_output_buffer(const char *input) {
 size_t len = strlen(input) + 10;
 char *buffer = (char *)calloc(len, sizeof(char));
 strcat(buffer, "[LOG]: ");
 strcat(buffer, input);
 return buffer;}
```

(b) Unsafe Memory Allocation

```c
int send_request(struct http_client *client, const char *payload) {
 int ret = http_send(client, payload);
 if (ret < 0) {
  fprintf(stderr, "Failed to send request\n");}
 log_transaction(client->response_code);
 return 0;}
```

(c) Logging Without Halting

```c
static void kvp_acquire_lock(int pool){
 struct flock fl = {F_WRLCK, SEEK_SET, 0, 0, 0};
 fl.l_pid = getpid();
 if (fcntl(kvp_file_info[pool].fd, F_SETLKW, &fl) == -1) {
  syslog(LOG_ERR, "Failed to acquire the lock pool: %d", pool);
  exit(EXIT_FAILURE);}}
```

(d) Missing Bounds Check

```c
void update_user_profile(struct user *u, const char *new_name) {
 if (u->profile_loaded) {
  strcpy(u->name, new_name);
  u->last_updated = time(NULL);}}
```

(e) Race Condition

Figure 1: Illustrative Examples of Common Risky Heuristic Patterns Utilized in HYDRA.

(1) **Input:** A set of fixed (patched) functions written in C, collected from a publicly available vulnerability patch dataset, BigVul.
(2) **Heuristic Feature Extraction Module:** Patched functions from input is parsed here using pattern-matching rules (e.g. regex) to detect the presence of known insecure coding practices (e.g. race condition, unsafe memory allocation) that leads to future zero-day attacks.
(3) **Semantic Embedding Module:** Simultaneously, the input patched functions are also passed to the pre-trained GraphCodeBERT for tokenization, which produces a high dimensional representation of the function's semantics based on both code tokens and implicit data-flow-graphs (DFG).
(4) **Mapping Alignment:** The corresponding heuristic vector of each functions is paired with its GraphCodeBERT embedding to form a training tuple (Embedding → heuristic vector).
(5) **Inference Module:** This module fuses the 768-dimensional GraphCodeBERT embedding with a 5 heuristic vector during training, forming a 773-dimensional input that is passed through a Variational Autoencoder (VAE) to learn compact latent representations. During testing, only the 768-dimensional embeddings are used to project unseen functions into the same latent space.
(6) **Clustering Module:** This module applies K-Means clustering on the latent representations learned by the VAE to group functions into vulnerability clusters and assign labels based on the proximity to heuristic traits or None regions.

Let's dig deeper into (2), (3), (4) and (5) modules.

3.2.1 Heuristic Feature Encoding. To capture interpretable, domain specific vulnerability signals, HYDRA incorporates a handcrafted heuristic extraction module implemented in python. This component is grounded in empirical security research and established CWE categorizations, particularly targeting vulnerability root causes frequently encountered in post-patch software. We design the following high impact binary vulnerability signatures:

- H_1: Missing null pointer check (→ may lead to CWE-476).
- H_2: Race condition check due to lack of synchronization (→ CWE-362).
- H_3: Missing bounds check on buffer/data (→ CWE-119, CWE- 120).
- H_4: Unsafe memory allocation check for $malloc/calloc/realloc$ failures (→ CWE-690).
- H_5: Logging errors without control flow blocking check (e.g. missing return or exit) (→ CWE-390, CWE-703).
- **None**: Represents functions with no heuristic match (benign functions or possibility to have unknown vulnerability). Not provided in input step.

For each patched function, we extract features using handcrafted regular expressions. where each function emits a heuristic match vector:

$$V_h \in \{0, 1\}^5$$

where each bit in the vector indicates the presence (1) or absence (0) of a specific vulnerability signature. For example $V_h = [1, 0, 0, 1, 0]$ indicates that the function both logs errors unsafely and lacks a null pointer check. This feature vector serves a dual purpose: **(1).** providing interpretable evidence that complements black-box deep learning predictions. **(2).** serving as an auxiliary signal to improve recall on semantically simple yet security critical cases often missed by deep models.

3.2.2 Embedding Extraction with GraphCodeBERT. While the heuristic extractor flags known vulnerability patterns via regex, it cannot capture latent or semantically obfuscated flaws. To address this, HYDRA integrates GraphCodeBERT [15], a pre-trained transformer

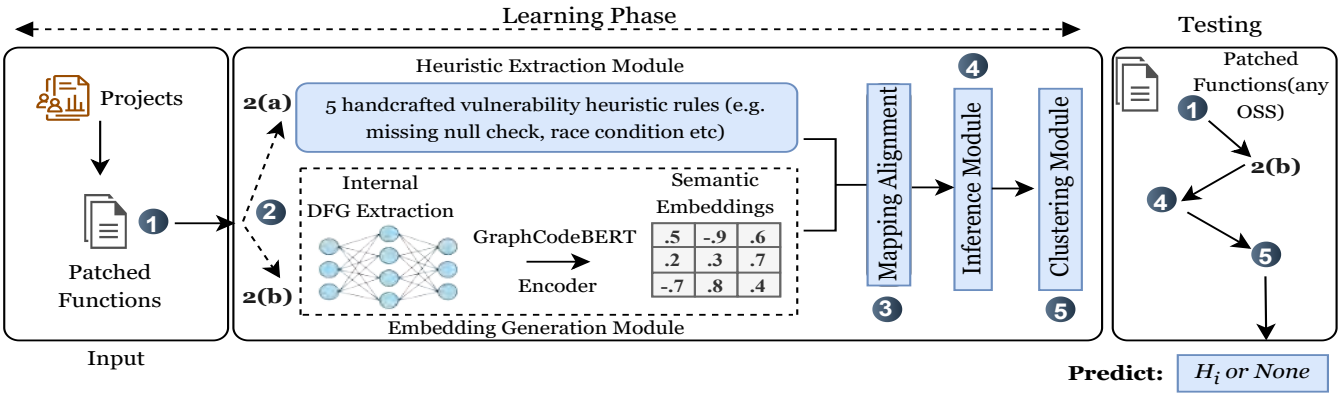

Figure 2: The proposed architecture of HYDRA.

encoder tailored for source code. It generates deep contextual embeddings of patched functions, enabling the model to generalize beyond surface syntax (*code syntax only*) and detect subtle indicators of risky logic or incomplete fixes. GraphCodeBERT leverages both token sequences and data flow graphs (DFGs) to learn a rich, compositional representation of program behavior.

Let f_i denote a preprocessed function in C, drawn from the patched function samples $F = \{f_1, f_2, ...f_n\}$. Each function f_i is tokenized into sequence of subword tokens (T):

$$T_i = \{t_{i1}, t_{i2}, ...t_{imi}\},\ t_{ij} \in \sum$$

where $\sum$ is the vocabulary space, and t_{ij} is the j-th token of the i-th function. Simultaneously, GraphCodeBERT constructs associated data flow graph:

$$G_i = \{v_i, \varepsilon_i\},\ v_i \subseteq T_i$$

where nodes v_i represents tokens that read/write shared variables, and edges $\varepsilon_i \subseteq v_i$ x v_i model variable dependencies. GraphCodeBERT then applies a dual learning objective to jointly optimize: **(1).** *Masked Language Modeling (MLM):* Predicting masked tokens from surrounding context. and **(2).** *Edge Prediction (EP):* Predicting existence of data flow edges between token pairs.

Through these objectives, a function f_i is encoded into a high-dimensional vector space:

$$E_s^{(i)} = GraphCodeBERT(T_i, G_i) \in R^d,\ d = 768$$

Here, d is dimensional real-valued vector space and R is the mathematical notation for a d-dimensional vector of real numbers. Each $E_s^{(i)}$ serves as a semantic embedding that captures structural, control-flow, and data-flow properties of function beyond lexical tokens.

We denote the complete embedding set across all samples as:

$$\epsilon_s = \{E_s^1, E_s^2, ...E_s^n\}$$

These embeddings are passed as real valued inputs into the downstream hybrid model alongside binary heuristic vectors (see Section 3.2.3). Empirically, we observe that GraphCodeBERT embeddings tend to organize functions into distinct semantic regions, where structurally or logically risky code patterns become separable in the high-dimensional embedding space.

In HYDRA, GraphCodeBERT is used as a frozen encoder, no parameter updates are performed ensuring reproducibility and leveraging knowledge distilled from CodeSearchNet and CodeXGLUE during pretraining.

3.2.3 Mapping Alignment and Inference Module. To unify symbolic vulnerability indicators with deep semantic representations, HYDRA employs a hybrid learning mechanism based on *early fusion*, where heuristically derived features and pretrained code embeddings are concatenated into a single latent feature space. This formulation supports both explainability and generalization during vulnerability inference.

Let us define:

$$V_h(x) = [v_1, v_2, \ldots, v_5], \text{where } v_i \in 0, 1$$

- $x \in C$ denotes a patched C function.
- $V_h(x) \in 0, 1^5$ is a binary vector indicating the presence of each heuristic pattern (e.g., $v_1 = 1$ if a null check is missing).
- $E_s(x) \in \mathbb{R}^{768}$ is the semantic embedding of x obtained using GraphCodeBERT.

The fused representation is defined as:

$$z(x) = \text{Concat}(V_h(x), E_s(x)) \in \mathbb{R}^{773}$$

To learn more compact latent vulnerability structure, we use a Variational Autoencoder (VAE) to project $z(x)$ into a compact latent space.

Clustering Module. We then apply K-Means clustering over the learned representations to group functions with similar vulnerability characteristics and assign risk labels from the set $\mathsf{H}_1, \mathsf{H}_2, \ldots, \mathsf{H}_5, \mathsf{None}$ based on cluster level heuristic prevalence.

II. Testing Phase. In test phase, for an unseen patched function X_{test}: **(1).** If $argmax\ f_0(L) = None$, HYDRA interprets the fixed function as either secure so far or as potentially containing an unknown (zero-day) vulnerability, (i.e., no known heuristic pattern is detected). **(2).** Otherwise, if the output corresponds to one of the five defined heuristic rules, HYDRA identifies the patched function as still containing latent vulnerability indicators such as "missing null check" thus flagging it as a potential zero-day vulnerability with explainable context. In this phase, only the unseen patched source functions from OSS is passed through the DL model pipeline to predict the future latent vulnerabilities from the patched functions.

Algorithm 1: HYDRA: Hybrid Deep Representation Architecture for Vulnerability **Prediction** on Patched Functions.

Input: Patched function corpus $\mathcal{F} = \{f_1, f_2, \ldots, f_n\}$;
Pre-trained GraphCodeBERT model G;
Heuristic rule set $\mathcal{H} = \{h_1, h_2, \ldots, h_5\}$
Output: Risk label vector $\mathcal{V} = \{v_1, v_2, \ldots, v_n\}$, where $v_i \in \mathcal{H} \cup \{\text{None}\}$
1 **foreach** $f_i \in \mathcal{F}$ **do**
2 Apply all $h_j \in \mathcal{H}$ to f_i via regex pattern matching;
3 Construct heuristic vector $H_i \in \{0, 1\}^5$;
4 Encode f_i using GraphCodeBERT:;
5 $E_i \leftarrow G(f_i)$, where $E_i \in \mathbb{R}^{768}$;
6 Concatenate: $Z_i = [E_i \parallel H_i] \in \mathbb{R}^{773}$;
7 Train a Variational Autoencoder (VAE) on $\{Z_1, Z_2, \ldots, Z_n\}$ to learn compact and continuous latent representations $\{L_1, L_2, \ldots, L_n\}$;
8 Cluster $\{L_i\}$ using K-Means to assign cluster labels $\{v_1, v_2, \ldots, v_n\}$, where $v_i \in \mathcal{H} \cup \{\text{None}\}$;
9 **return** $\mathcal{V}$

The overall HYDRA model **prediction** is described in **Algorithm** 1. Here each function f_i is first analyzed using five heuristic rules (lines 2–3). GraphCodeBERT then encodes the function into a 768-dimensional embedding E_i (lines 4–5), which is concatenated with the 5-dimensional heuristic vector H_i to form a fused 773-dimensional vector Z_i (line 6). This vector is passed through a Variational Autoencoder (VAE) to generate latent representations L_i (line 7). Finally, K-Means clustering groups similar functions, and heuristic-based labels including None are assigned based on latent structure (lines 8–9).

4 EVALUATION OF HYDRA

We evaluate the effectiveness of HYDRA based on following research questions:

- **RQ1:** Can HYDRA predict latent risky patterns (heuristic rules) in patched functions missed by symbolic or deep models that may lead to zero-day vulnerabilities?
- **RQ2:** How do HYDRA's components influence the quality of learned representations for clustering patched functions?
- **RQ3:** Can HYDRA anticipate novel or previously unseen vulnerability patterns not explicitly covered during training?

4.1 Experimental Setup

Implementation. The deep learning components (including the integration of GraphCodeBERT and classifier models) are implemented using the PyTorch (v2.7.1) [2] and the HuggingFace Transformers (v4.37) library [5]. The heuristic rule engine for feature extraction is implemented with custom regex modules and Python's built-in AST parser for syntactic verification. All experiments are conducted on a multi-core Linux server equipped with Intel Core i7-12700F CPU (3.0GHz × 8 cores), and NVIDIA RTX 4070 GPU (12GB VRAM), and 16GB of RAM, running Kali 2023.2 with Linux kernel version 6.1.

HYDRA and Baseline Models. To the best of our knowledge, there are no existing baseline models in unsupervised settings that specifically designed for post-patch vulnerability **prediction** at the time of developing this work. Hence, to assess HYDRA's effectiveness in this unsupervised setting for vulnerability prediction, we construct multiple baseline design variant models, detailed as follows:

[M1] **Regex + K-means**: A naive most basic baseline model using only handcrafted regex-based heuristic rules, where K-means is applied to cluster functions based solely on their binary rule presence.

[M2] **GraphCodeBERT + K-means**: We extract semantic embeddings of the functions using GraphCodeBERT and apply K-means to identify high density regions of potential vulnerability patterns.

[M3] **Regex + GraphCodeBERT + K-means**: Combines the regex-based heuristic rule vectors with GraphCodeBERT embeddings via early fusion. The concatenated representation is directly clustered using K-means to uncover latent structure informed by both symbolic and semantic signals.

[HYDRA] **Regex + GraphCodeBERT + VAE + K-means**: Our proposed architecture uses a Variational Autoencoder (VAE) trained on early fused GraphCodeBERT embeddings and heuristic rule vectors. At inference time, only the embeddings are provided to the VAE to project functions into a latent space, where clustering is applied to reveal potential risk patterns.

Dataset Construction. We construct our dataset using Big-Vul [12], a large scale vulnerability dataset based on the National Vulnerability Database (NVD) [1], which provides real-world vulnerable and fixed code pairs. Specifically, we extract 20,451 patched C functions from the Linux project for training our model, sourced from CVE linked repositories [11]. These functions are treated as the fixed versions of previously vulnerable code making them ideal for studying residual risks and latent vulnerabilities that may still lead to future zero-day exploits. To evaluate generalization, we test our models on patched functions from three diverse codebases: Chrome (16,387 functions), Android (2,322 functions), and ImageMagick (1,703 functions). We use these projects due to their diversity in application domains, code complexity, and real-world vulnerability history.

Evaluation Metrics. Given the unsupervised nature of HYDRA, to assess the effectiveness of it's latent representations in forming semantically meaningful clusters, we evaluate clustering quality using three standard unsupervised metrics Silhouette Score [22], Calinski-Harabasz Index (CHI) [20], and Davies-Bouldin Index (DBI) [21]. Each metric quantifies the balance between intra-cluster cohesion and inter-cluster separation. The ranges for these scores are $[-1, 1]$, $[0, \infty)$ and $[0, \infty)$ respectively. Higher values for Silhouette and CHI indicate better clustering, while lower DBI values indicate better cluster separation and compactness.

4.2 RQ1: Predicting Latent Risky Heuristic Patterns by HYDRA

Method. To assess HYDRA's effectiveness in identifying latent risks in patched functions particularly those missed by symbolic (regex) or semantic (GraphCodeBERT) methods, we evaluate heuristic **prediction** performance across three model variants: M1, M3,

and HYDRA. The model M2 (GraphCodeBERT + KMeans) is not included here, as it produces purely semantic representations without any connection to the heuristic labels. HYDRA is explicitly designed to align both symbolic and semantic cues with risk patterns. The models are trained to associate these representations with predefined heuristic labels H_1 to H_5. During testing, they predict the presence of these risk patterns in patched functions drawn from three real-world projects: Chrome, Android, and ImageMagick. Detailed results for Chrome are shown in Table 1, with full cross-project summaries provided in Table 2.

Results. Table 1 summarizes the heuristic rule **prediction** match counts for the Chrome test dataset across three model variants - M1, M3, and HYDRA - illustrating how total predicted matches are computed for a single project. While HYDRA detects fewer total matches (788) than M1 (1126) and M3 (921), this reduction reflects its focus on learning more compact and generalizable representations. For example, HYDRA captures a comparable number of H_2 matches (292), yet avoids potential over-prediction in H_1 and H_3, which are more susceptible to superficial matches in simpler models. Although M1 yields the highest count, its reliance on regex alone increases the likelihood of redundant or noisy predictions. HYDRA, by integrating symbolic and semantic features through a VAE-based encoder, emphasizes representational fidelity, producing more selective yet interpretable clusters. These results suggest that HYDRA enables conservative but meaningful heuristic alignment, improving generalization and prediction of residual risks in patched function. Notably, H_4 and H_5 — unsafe memory allocation and unsafe logging were not observed in Chrome. This likely reflects their rarity in real-world patches, where developers prioritize immediately visible issues. Additionally, such patterns often require deeper semantic inference or runtime context, which may not be evident in static code alone.

Table 1: Heuristic rule prediction counts by model variants (M1, M3, HYDRA) on the Chrome test project of patched functions.

Heuristic Rules	M1	M3	HYDRA
H_1	645	588	460
H_2	292	278	292
H_3	189	55	36
H_4	0	0	0
H_5	0	0	0
Total Matched Heuristic	1126	921	788

Cross project analysis. Table 2 presents matched heuristic rule counts and their corresponding percentages across Chrome, Android, and ImageMagick for various model variants. The hybrid HYDRA model consistently yields the lowest match rates 4.80% in Chrome, 11.1% in Android, and 16.03% in ImageMagick indicating stronger filtering of false positives while still capturing meaningful risk patterns. In contrast, models like M1 and M3 report significantly higher match rates (e.g., M1 shows 6.87% in Chrome and 22.19% in ImageMagick), suggesting over-matching due to weaker generalization. HYDRA's integration of symbolic rules with latent code representations improves precision and enables more refined post-patch vulnerability **prediction** across diverse codebases. Its performance on both large-scale projects (e.g., Chrome) and smaller ones (e.g., ImageMagick) highlights the robustness and adaptability of the hybrid approach.

Table 2: Heuristic rule prediction summary for Chrome, Android, and ImageMagick across model variants. Values indicate total matches and corresponding percentages.

Model	Chrome	Android	ImageMagick
M1	1126 (6.87%)	411 (17.7%)	378 (22.19%)
M3	921 (5.62%)	304 (13.1%)	351 (20.61%)
HYDRA	**788 (4.80%)**	**257 (11.1%)**	**273 (16.03%)**

4.3 RQ2: Impact of HYDRA's Components on Clustering Patched Functions

Method. While RQ1 focuses on model accuracy in predicting known risky patterns, RQ2 shifts focus to understanding how individual components of HYDRA influence the quality of learned internal representations of patched functions used for vulnerability inference. To assess this, we examine how well the function embeddings from each model variant form meaningful clusters in an unsupervised setting. This analysis will help us assess whether HYDRA captures meaningful structure in patched function without relying on explicit vulnerability labels. If distinct clusters emerge, it suggests HYDRA has learned to organize functions based on latent risk patterns even when no known bug type is specified. Together, RQ1 and RQ2 demonstrate HYDRA's ability to both predict and group subtle post-patch security flaws. To evaluate HYDRA's overall performance and understand the role of each model component, we compare baseline variants and HYDRA using the unsupervised clustering metrics introduced in Section 4.1, which assess structural quality (compactness and separation) independently of labeled data. This comparison reveals the incremental contribution of symbolic heuristics, pretrained embeddings, and VAE-based representation learning in shaping the internal structure of the learned space.

Results: Table 3 presents the unsupervised clustering quality metrics such as Silhouette Score (↑), CH Index (↑), and DB Index (↓) for all models across the Chrome, Android, and ImageMagick test projects. HYDRA consistently outperforms all other models in unsupervised clustering quality across all projects, achieving the highest Silhouette scores, maximum CHI values and the lowest DBI scores. These results indicate that HYDRA forms well-separated, cohesive clusters in latent space, even without heuristic supervision during inference. Although M1 (regex + K-Means) achieves perfect clustering scores, this is due to its trivial binary input, offering no real semantic insight or generalization. M2, using only GraphCodeBERT embeddings, performs better on clustering metrics but forms less meaningful clusters. In contrast, M3 (regex + embeddings) captures richer vulnerability structure through symbolic semantic fusion. HYDRA surpasses all models, with its VAE based encoding producing compact, well-separated, and semantically meaningful clusters enabling the **prediction** of subtle post-patch risks that simpler models overlook.

Interestingly, all embedding based models (M2, M3, HYDRA) consistently form two semantic clusters across projects, each dominant by `None` labeled functions. Figure 3 shows the semantic clustering results derived from HYDRA, which illustrates distribution of `None` labeled patched functions across semantic embedding clusters for each project. While `None` cases dominate these clusters, they consistently align with specific heuristic types (H), suggesting latent

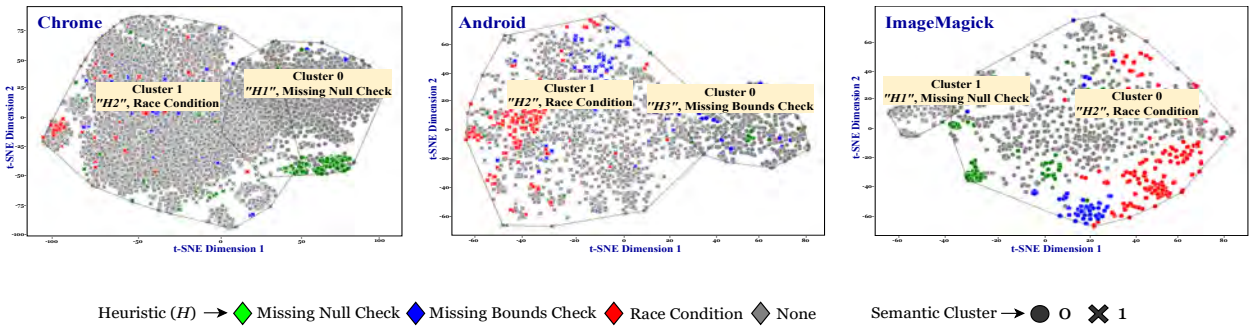

Figure 3: Distribution of None labeled patched functions across semantic embedding clusters for each project by HYDRA. Although None cases dominate each cluster, they tend to align with a specific heuristic type H (annotated), indicating latent similarity to known vulnerability patterns within that semantic cluster.

Table 3: Unsupervised clustering quality of models across all projects using Silhouette, CHI and DBI metrics.

Model	Silhouette (↑)	CHI (↑)	DBI (↓)
Chrome			
M1	1.00	1.00	0.00
M2	0.15	3405.30	2.05
M3	0.14	3388.33	2.06
HYDRA	**0.54**	**21940.15**	**0.81**
Android			
M1	0.99	1.00	0.00
M2	0.16	376.34	2.43
M3	0.16	371.86	2.45
HYDRA	**0.61**	**3802.89**	**0.72**
ImageMagick			
M1	0.99	1.00	0.00
M2	0.16	283.36	2.37
M3	0.15	278.94	2.40
HYDRA	**0.68**	**1993.26**	**0.66**

similarities to known vulnerability pattern within each semantic cluster. This suggests that most of the unflagged functions often share latent structural traits like control flow or data usage and this behavior is likely influenced by the GraphCodeBERT embeddings which encode rich semantic and syntactic relationship. HYDRA, in particular, surfaces these high risk, anomalous patterns that may escape rule based **prediction** and pose zero-day risks. In contrast, M1 fails to form such clusters, underscoring the limitations of purely symbolic methods and the representational strength of HYDRA's hybrid approach.

Effectiveness: HYDRA outperforms all variants across clustering metrics by combining Regex rules, GraphCodeBERT embeddings, and VAE-based latent learning, yielding more coherent and semantically meaningful clusters.To further analyze this improvement, we examine HYDRA's VAE reconstruction loss. As depicted in Figure 4, the model converges rapidly within the first few epochs and maintains stable error reduction across 200 epochs. The lowest validation loss occurs at epoch 186, approaching zero. This stability confirms that HYDRA's VAE effectively learns a low dimensional latent space that retains both heuristic and semantic structure. This latent alignment, combined with downstream K-means clustering, enables HYDRA to deliver strong **prediction** performance and structural generalization, as evidenced in RQ1 and RQ2.

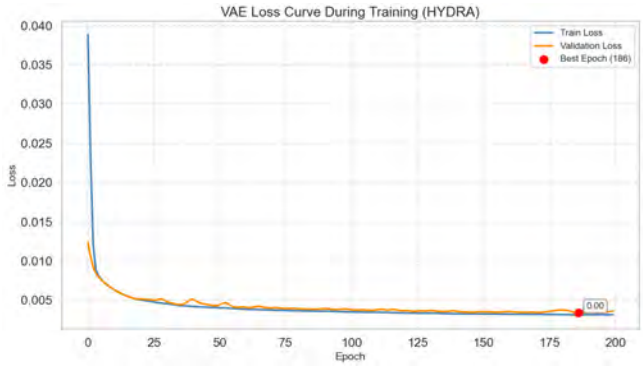

Figure 4: VAE reconstruction loss during HYDRA training. Both training and validation losses converge smoothly, with best validation performance at epoch 186.

4.4 RQ3: HYDRA's Capacity to Identify Novel/Unseen Risk Patterns

Method. To evaluate HYDRA's ability to generalize beyond predefined heuristic rules, we focus on its predictions labeled as None cases not matching any of the five heuristics. A high proportion of such cases in every cluster across all the projects as seen in Figure 3 suggests HYDRA's potential to uncover latent risks. Building on RQ2, which showed that HYDRA forms well-separated, meaningful clusters, RQ3 examines how these None cases tend to align with a specific heuristic type H of known vulnerabilities. This suggests HYDRA can uncover overlooked or emerging vulnerability patterns that may signal future zero-day threats.

Table 4: Prediction of None labeled samples from all three model variants. Values indicate total counts and corresponding percentages.

Model	Chrome	Android	ImageMagick
M1	0	0	0
M3	15446 (94.3%)	2018 (86.9%)	1352 (79.4%)
HYDRA	**15599** (95.2%)	**2065** (88.9%)	**1430** (83.97%)

Results: Table 4 reports the proportion of samples labeled as None across the three model variants. As expected, the regex only model

(M1) produces no None predictions due to its deterministic design it can only match explicitly defined patterns, limiting its ability to surface novel or ambiguous risks, hence 0 count. In contrast, the latent space models M3 and HYDRA predict substantial proportions of None labels, with HYDRA reaching 95.2% in Chrome, 88.9% in Android, and 83.97% in ImageMagick. This reflects HYDRA's greater generalization capacity beyond handcrafted rules.

To assess the plausibility of these None predictions, we analyze their distribution in HYDRA's latent space and identify the known heuristic it aligns the most with in each of the two distinct semantic clusters 0 and 1. Table 5 highlights the most probabilistically aligned/dominant heuristic type H_A in each of the clusters for None labeled samples across Chrome, Android, and ImageMagick based on the highest softmax confidence scores produced by HYDRA's heuristic **prediction** head. For example, in the Chrome project, most None cases align with two dominant heuristics: 934 samples from cluster 0 map to H_1 (CWE-476: NULL pointer dereference), and 530 samples from cluster 1 resemble H_2 (CWE-362: Thread non-synchronization). Similar dual-pattern alignments emerge in Android (H_2, H_3) and ImageMagick (H_1, H_2). These probabilistic associations arise from HYDRA's latent space. Furthermore we manually inspected a representative subset of None predicted functions with two independent software engineers (K = 0.89), observing that HYDRA's latent space probabilistically clusters structurally and semantically similar functions near regions associated with known heuristics, even without explicit matches. Rather than hard relabeling, this behavior offers a soft signal of latent risk, surfacing functions potentially indicative of emerging or uncharacterized vulnerabilities. These associations show HYDRA's potential to surface risky code patterns missed by symbolic methods, aiding zero-day vulnerability **prediction**. By aligning with heuristic and CWE contexts, HYDRA enhances interpretability and supports informed triage even without explicit rule matches.

Table 5: Most dominant heuristic aligned types (H_A) for None labeled samples in Cluster 0 and Cluster 1 across Chrome, Android, and ImageMagick, as positioned in HYDRA's latent space.

Project	Cluster(H_A)	None Count	CWE Match	Interpretation
Chrome	0 (H_1)	934	CWE-476	NULL dereference
	1 (H_2)	530	CWE-362	Thread non-synchronization
Android	0 (H_2)	157	CWE-362	Thread non-synchronization
	1 (H_3)	65	CWE-119	Buffer overflow
ImageMagick	0 (H_1)	29	CWE-476	NULL dereference
	1 (H_2)	106	CWE-362	Thread non-synchronization

5 DISCUSSION

To further validate HYDRA's practical impact in the context of software security testing, we analyzed patched functions by identifying risky patterns through two real-world case studies from the Chrome and ImageMagick software projects.

ImageMagick. A representative case from the ImageMagick project (Figure 5) shows that, despite patching, subtle issues like unchecked null pointer usage can persist posing latent risks for future zero-day vulnerabilities. In this case, the wrapper function `ClipImage` uses the input pointer `image` (marked yellow, line 1 and 4) without verifying its validity, risking a null dereference. HYDRA identifies such post-patch flaws by combining symbolic heuristics with learned code representations, revealing incomplete fixes often missed by manual review or traditional scanners.

Possible Corrected Version: A safeguard is added (marked cyan, lines 2–3 from Figure 5) to ensure the pointer is not null before use, effectively mitigating the vulnerability. Such missing null checks can result in undefined behavior, application crashes, or denial-of-service conditions when dereferenced improperly. This issue corresponds to CWE-476 (NULL Pointer Dereference) and poses significant risks in systems processing untrusted inputs, potentially leading to memory corruption or enabling exploit chains.

```
1 MagickExport MagickBooleanType ClipImage(Image *image){
2  if (image == NULL)
3    return MagickFalse;  // or raise an error/log appropriately
4  return ClipImagePath(image, "#1", MagickTrue);}
```

Figure 5: Example from ImageMagick: HYDRA flags the `ClipImage` wrapper function using heuristic H1 (missing null check).

Chrome. Another example from the Chrome project (see Figure 6) shows a function labeled as None by HYDRA. Although the expression: `pending_entry_index_=current_index-1` suggests a potential underflow risk, HYDRA's H_3 heuristic did not trigger. This is because H_3 targets recognizable patterns of unsafe index use, such as: (1) unvalidated index access (e.g.`array[i]`, `vector.at(i)`, `buffer[i]`), and (2) index comparisons (e.g., $i < size$, $i >= 0$). Since this case involves a simple assignment without dereference or container access, the symbolic rule was not matched. Though `GoBack()` did not match HYDRA's symbolic H_3 heuristic, its latent embedding placed it near H_3 labeled samples due to structural similarities specifically, decrementing an index without bounds checking and using it in a downstream state transition (marked yellow, line 5 and 9).

Possible Corrected Version: There has been added a bounds check (marked cyan, lines 6–7, in Figure 6) such as `if(current_index <= 0)`, preventing underflow and mitigating this latent vulnerability. HYDRA's learned representation correctly associates this with H_3 style risks, highlighting its ability to flag latent issues beyond explicit rule matches. This unsafe arithmetic aligns with CWE-1285: Improper Validation of Specified Index, where missing lower bound checks can lead to invalid states or out-of-bounds behavior. Such subtle flaws often evade conventional testing and, under certain conditions, may be exploitable as zero-day vulnerabilities.

These case studies highlight HYDRA's ability to **predict** latent both known and unknown vulnerabilities by combining symbolic rules with deep code representations. This fusion enables HYDRA to surface subtle post-patch flaws that symbolic methods miss, making it a valuable tool for a robust software security testing leveraging zero-day risk **prediction**.

6 THREATS TO VALIDITY

Internal Validity. A key consideration for internal validity is the design of the heuristic labeling process. HYDRA is trained using five manually defined rules derived from regular expressions over

```
1 void NavigationController::GoBack(){
2  if(!CanGoBack()){
3    NOTREACHED();
4    return;}
5  int current_index = GetCurrentEntryIndex();
6  if (current_index <= 0) {
7    return;}
8  DiscardNonCommittedEntries();
9  pending_entry_index_ = current_index - 1;
10 NavigateToPendingEntry(false);}
```

Figure 6: Example from Chrome: `GoBack` function is marked as `None` by HYDRA but semantically aligns with heuristic H3 (missing bounds check).

preprocessed source code. While these rules are based on widely accepted and frequently observed vulnerability patterns (e.g., missing null or bounds checks) in real-world CVEs, manual construction may introduce inconsistencies or overlook edge cases, potentially introducing noise into the training set. Importantly, HYDRA does not label None classified functions as definitively safe or vulnerable. Instead, it learns latent representations that capture semantic and structural similarity. When None samples appear near known risky clusters, we interpret this as a soft signal of latent risk, informed by qualitative review not as hard relabeling. This ability to surface structurally similar but heuristically silent code offers practical utility in vulnerability triage and zero-day threat anticipation. Due to the lack of ground **truth labels** for post-patch vulnerability assessment, we did not perform direct comparative analysis and compute conventional metrics (e.g., F1, AUC, MCC), focusing instead on unsupervised vulnerability **prediction** and heuristic alignment.

External Validity: Our dataset consists of patched (non-vulnerable) C/C++ functions sourced from Linux-based repositories in the BigVul dataset. While C/C++ are widely studied in software security, HYDRA has not yet been evaluated on other languages like Java, Python, or Rust, limiting generalizability across different programming paradigms. HYDRA produces both heuristic class predictions and a None class, indicating that a function does not match any predefined rule potentially signaling a zero-day or novel pattern. However, without a definitive ground truth for None cases, this interpretation remains probabilistic. While it aligns with the goal of uncovering latent vulnerabilities, quantitative evaluation of None predictions is challenging. These cases were partially validated through expert review and manual inspection. Future work will explore stronger validation signals, such as fuzzing based analysis or synthetic vulnerability injection.

7 RELATED WORK

Vulnerability Detection in Source Code. Extensive research has focused on deep learning models trained on large vulnerability datasets to detect insecure code. Approaches like VulDeePecker [46], DeepCVA [19], and LineVul [13] use token-level or structured embeddings to classify code as vulnerable or not. More recent methods such as hybrid models like IVDetect [23] and ReVeal [17], and graph based models like Devign [44] leverage commit-level changes and control/data flow representations to improve generalization. However, most of these rely on explicitly labeled buggy code, limiting their ability to detect latent or residual risks in previously patched function an area that HYDRA is specifically designed to address.

Heuristic and Static Rule Based Detection. Conventional tools like Cppcheck [26] and Flawfinder [42] rely on manually crafted rules or regex to detect issues such as use-after-free, buffer overflows, and missing null checks. While interpretable, these tools often lack precision and fail to generalize to subtle, latent vulnerabilities in real-world code. In contrast, HYDRA integrates such heuristics as structured feature vectors, guiding deep representations toward known vulnerability patterns.

Patch Analysis and Silent Fix Studies. Recent efforts like GraphSPD [39], PatchRNN [41], and GRAPE [16] have begun analyzing post-fix code to detect silent fixes and assess patch security. These methods often rely on change classification or learning pre- and post-patch code representations. However, they typically assume the patch is correct or focus on binary classification of patch types. In contrast, HYDRA evaluates previously patched function to uncover residual risks, identifying cases where fixes are incomplete or incorrectly applied.

Hybrid and Casual Models in Vulnerability Learning. Hybrid systems like DeepDFA [36] and CausalVul [33] combine symbolic reasoning with deep learning to improve generalization beyond spurious correlations. HYDRA builds on this direction by integrating human interpretable heuristics with deep semantic representations from GraphCodeBERT, forming a unified architecture that balances precision and explainability for detecting silent vulnerabilities in post-fix code.

Zero-Day and Out-of-Distribution (OOD) Prediction. To combat zero-day threats, approaches like NERO [8], UL-VAE [6], and open-set intrusion detection systems use anomaly or out-of-distribution (OOD) detection, typically over binary data or traffic logs via autoencoders or meta-learning. In contrast, HYDRA is tailored for source code and requires neither CVE labels nor runtime triggers enabling probabilistic forecasting of latent zero-day risks directly from patched functions.

8 CONCLUSION & FUTURE WORK

In this work, we introduce HYDRA, a hybrid vulnerability analysis framework designed to identify residual risky patterns in patched functions potential indicators of zero-day vulnerabilities. HYDRA leverages a dual layered approach: combining handcrafted heuristic rules (regex-based) with deep code representations learned via GraphCodeBERT and a Variational Autoencoder. This integration allows the system to surface latent, high risk patterns that persist even after developers issue security patches patterns that might otherwise evade detection using traditional static or learned techniques in isolation. Our empirical evaluation across three real-world projects Chrome, Android, and ImageMagick shows that HYDRA reduces overmatching compared to heuristic only baselines while retaining the ability to flag vulnerable code segments that may still harbor security weaknesses. The cross project generalization capabilities highlight HYDRA's effectiveness in spotting common coding flaws that survive across diverse codebases. These findings underscore the pressing need to revisit post patch code with a deeper,

hybrid analysis lens, as even patched functions can remain susceptible to exploitation a characteristic often seen in zero-day scenarios. As future work, we plan to expand HYDRA by incorporating a richer set of heuristics and evaluating on diverse datasets across languages and domains, including android and cloud systems, to better assess post-patch zero-day risks while further integrating it into automated security testing pipelines to improve post-patch zero-day vulnerability assessment and validation.

References

[1] [n. d.]. *National Vulnerability Database (NVD).* https://nvd.nist.gov Accessed March 30, 2025.

[2] [n. d.]. *PyTorch.* https://pytorch.org/ Accessed March 15, 2025.

[3] [n. d.]. *Solarwinds.* https://www.solarwinds.com/orion-platform Accessed March 30, 2025.

[4] [n. d.]. *Stuxnet.* https://www.malwarebytes.com/stuxnet Accessed March 29, 2025.

[5] [n. d.]. *Transformers.* https://pypi.org/project/transformers/4.37.0/ Accessed March 15, 2025.

[6] Namrata Govind Ambekar and Surmila Thokchom. 2024. UL-VAE: An Unsupervised Learning Approach for Zero-day Malware Detection Using Variational Autoencoder. In *2024 International Conference on Computational Intelligence and Network Systems (CINS)*. 1–7. doi:10.1109/CINS63881.2024.10864450

[7] Leyla Bilge and Tudor Dumitraş. 2012. Before we knew it: an empirical study of zero-day attacks in the real world. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security* (Raleigh, North Carolina, USA) *(CCS '12)*. Association for Computing Machinery, New York, NY, USA, 833–844. doi:10.1145/2382196.2382284

[8] Jesús F. Cevallos M., Alessandra Rizzardi, Sabrina Sicari, and Alberto Coen Porisini. 2024. NERO: NEural algorithmic reasoning for zeRO-day attack detection in the IoT: A hybrid approach. *Computers & Security* 142 (2024), 103898. doi:10.1016/j.cose.2024.103898

[9] Haogang Chen, Yandong Mao, Xi Wang, Dong Zhou, Nickolai Zeldovich, and M. Frans Kaashoek. 2011. Linux kernel vulnerabilities: state-of-the-art defenses and open problems *(APSys '11)*. Association for Computing Machinery, New York, NY, USA, Article 5, 5 pages. doi:10.1145/2103799.2103805

[10] The MITRE Corporation. CWE Details. https://cwe.mitre.org/. Accessed March 12, 2025.

[11] The MITRE Corporation. CVE Details. https://www.cve.org/. Accessed March 12, 2025.

[12] Jiahao Fan, Yi Li, Shaohua Wang, and Tien N. Nguyen. 2020. A C/C++ Code Vulnerability Dataset with Code Changes and CVE Summaries. In *2020 IEEE/ACM 17th International Conference on Mining Software Repositories (MSR)*. 508–512. doi:10.1145/3379597.3387501

[13] 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* (Pittsburgh, Pennsylvania) *(MSR '22)*. Association for Computing Machinery, New York, NY, USA, 608–620. doi:10.1145/3524842.3528452

[14] Michael Fu, Chakkrit Tantithamthavorn, Trung Le, Van Nguyen, and Dinh Phung. 2022. VulRepair: a T5-based automated software vulnerability repair. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Singapore, Singapore) *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 935–947. doi:10.1145/3540250.3549098

[15] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. 2020. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366* (2020).

[16] Mei Han, Lulu Wang, Jianming Chang, Bixin Li, and Chunguang Zhang. 2024. Learning Graph-based Patch Representations for Identifying and Assessing Silent Vulnerability Fixes. In *2024 IEEE 35th International Symposium on Software Reliability Engineering (ISSRE)*. 120–131. doi:10.1109/ISSRE62328.2024.00022

[17] Ziniu Hu, Ahmet Iscen, Chen Sun, Zirui Wang, Kai-Wei Chang, Yizhou Sun, Cordelia Schmid, David A. Ross, and Alireza Fathi. 2023. REVEAL: Retrieval-Augmented Visual-Language Pre-Training with Multi-Source Multimodal Knowledge Memory. *CVPR* (2023).

[18] Alya Hannah Ahmad Kamal, Caryn Chuah Yi Yen, Gan Jia Hui, Pang Sze Ling, et al. 2020. Risk assessment, threat modeling and security testing in SDLC. *arXiv preprint arXiv:2012.07226* (2020).

[19] Triet Huynh Minh Le, David Hin, Roland Croft, and M. Ali Babar. 2022. DeepCVA: automated commit-level vulnerability assessment with deep multi-task learning. In *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering* (Melbourne, Australia) *(ASE '21)*. IEEE Press, 717–729. doi:10.1109/ASE51524.2021.9678622

[20] Scikit-learn Developers. sklearn.metrics.calinski_harabasz_score. https://scikit-learn.org/stable/modules/generated/sklearn.metrics.calinski_harabasz_score.html. Accessed July 14, 2025.

[21] Scikit-learn Developers. sklearn.metrics.davies_bouldin_score. https://scikit-learn.org/stable/modules/generated/sklearn.metrics.davies_bouldin_score.html. Accessed July 14, 2025.

[22] Scikit-learn Developers. sklearn.metrics.silhouette_score. https://scikit-learn.org/stable/modules/generated/sklearn.metrics.silhouette_score.html. Accessed July 14, 2025.

[23] Yi Li, Shaohua Wang, and Tien N. Nguyen. 2021. Vulnerability detection with fine-grained interpretations. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 292–303. doi:10.1145/3468264.3468597

[24] Yi Li, Aashish Yadavally, Jiaxing Zhang, Shaohua Wang, and Tien N. Nguyen. 2023. Commit-Level, Neural Vulnerability Detection and Assessment. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (San Francisco, CA, USA) *(ESEC/FSE 2023)*. Association for Computing Machinery, New York, NY, USA, 1024–1036. doi:10.1145/3611643.3616346

[25] Georgios Michail Makrakis, Constantinos Kolias, Georgios Kambourakis, Craig Rieger, and Jacob Benjamin. 2021. Vulnerabilities and attacks against industrial control systems and critical infrastructures. *arXiv preprint arXiv:2109.03945* (2021).

[26] Daniel Marjamaki. [n. d.]. https://cppcheck.sourceforge.io/. ([n. d.]). Accessed February 14, 2025.

[27] D Nandakumar, R Schiller, C Redino, K Choi, A Rahman, E Bowen, M Vucovich, J Nehila, M Weeks, and A Shaha. [n. d.]. Zero day threat detection using metric learning autoencoders (2022).

[28] Chao Ni, Xin Yin, Kaiwen Yang, Dehai Zhao, Zhenchang Xing, and Xin Xia. 2023. Distinguishing Look-Alike Innocent and Vulnerable Code by Subtle Semantic Representation Learning and Explanation. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (San Francisco, CA, USA) *(ESEC/FSE 2023)*. Association for Computing Machinery, New York, NY, USA, 1611–1622. doi:10.1145/3611643.3616358

[29] Yu Nong, Richard Fang, Guangbei Yi, Kunsong Zhao, Xiapu Luo, Feng Chen, and Haipeng Cai. 2024. VGX: Large-Scale Sample Generation for Boosting Learning-Based Software Vulnerability Analyses. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 149, 13 pages. doi:10.1145/3597503.3639116

[30] Yu Nong, Yuzhe Ou, Michael Pradel, Feng Chen, and Haipeng Cai. 2023. VULGEN: Realistic Vulnerability Generation Via Pattern Mining and Deep Learning. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 2527–2539. doi:10.1109/ICSE48619.2023.00211

[31] Hammond Pearce, Benjamin Tan, Baleegh Ahmad, Ramesh Karri, and Brendan Dolan-Gavitt. 2023. Examining Zero-Shot Vulnerability Repair with Large Language Models . In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 2339–2356. doi:10.1109/SP46215.2023.10179420

[32] Zeqing Qin, Yiwei Wu, and Lansheng Han. 2025. CLNX: Bridging Code and Natural Language for C/C++ Vulnerability-Contributing Commits Identification. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 39. 25047–25055.

[33] Md Mahbubur Rahman, Ira Ceka, Chengzhi Mao, Saikat Chakraborty, Baishakhi Ray, and Wei Le. 2024. Towards causal deep learning for vulnerability detection. In *Proceedings of the IEEE/ACM 46th international conference on software engineering*. 1–11.

[34] Yaman Roumani. 2021. Patching zero-day vulnerabilities: an empirical analysis. *Journal of Cybersecurity* 7, 1 (11 2021), tyab023. doi:10.1093/cybsec/tyab023 arXiv:https://academic.oup.com/cybersecurity/article-pdf/7/1/tyab023/41180532/tyab023.pdf

[35] Karuturi Sneha and Gowda M Malle. 2017. Research on software testing techniques and software automation testing tools. In *2017 International Conference on Energy, Communication, Data Analytics and Soft Computing (ICECDS)*. 77–81. doi:10.1109/ICECDS.2017.8389562

[36] Benjamin Steenhoek, Hongyang Gao, and Wei Le. 2024. Dataflow Analysis-Inspired Deep Learning for Efficient Vulnerability Detection. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 16, 13 pages. doi:10.1145/3597503.3623345

[37] Jiamou Sun, Zhenchang Xing, Qinghua Lu, Xiwei Xu, Liming Zhu, Thong Hoang, and Dehai Zhao. 2023. Silent Vulnerable Dependency Alert Prediction with Vulnerability Key Aspect Explanation. In *Proceedings of the 45th International Conference on Software Engineering* (Melbourne, Victoria, Australia) *(ICSE '23)*. IEEE Press, 970–982. doi:10.1109/ICSE48619.2023.00089

[38] Maneela Tuteja, Gaurav Dubey, et al. 2012. A research study on importance of testing and quality assurance in software development life cycle (SDLC) models. *International Journal of Soft Computing and Engineering (IJSCE)* 2, 3 (2012), 251–257.

[39] Shu Wang, Xinda Wang, Kun Sun, Sushil Jajodia, Haining Wang, and Qi Li. 2023. GraphSPD: Graph-Based Security Patch Detection with Enriched Code Semantics . In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 2409–2426. doi:10.1109/SP46215.2023.10179479

[40] Xinda Wang, Kun Sun, Archer Batcheller, and Sushil Jajodia. 2019. Detecting" 0-day" vulnerability: An empirical study of secret security patch in OSS. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 485–492.

[41] Xinda Wang, Shu Wang, Pengbin Feng, Kun Sun, Sushil Jajodia, Sanae Benchaaboun, and Frank Geck. 2021. PatchRNN: A Deep Learning-Based System for Security Patch Identification. In *MILCOM 2021 - 2021 IEEE Military Communications Conference (MILCOM)* (San Diego, CA, USA). IEEE Press, 595–600. doi:10.1109/MILCOM52596.2021.9652940

[42] David A. Wheeler. [n. d.]. https://dwheeler.com/flawfinder/. ([n. d.]). Accessed February 14, 2025.

[43] Alexander A. Zakharov and Kirill I. Gladkikh. 2024. Characteristics and Trends of Zero-Day Vulnerabilities in Open-Source Code. In *2024 International Russian Automation Conference (RusAutoCon)*. 498–502. doi:10.1109/RusAutoCon61949.2024.10694228

[44] 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*. Curran Associates Inc., Red Hook, NY, USA.

[45] Enrico Zio. 2016. Challenges in the vulnerability and risk analysis of critical infrastructures. *Reliability Engineering & System Safety* 152 (2016), 137–150. doi:10.1016/j.ress.2016.02.009

[46] Deqing Zou, Sujuan Wang, Shouhuai Xu, Zhen Li, and Hai Jin. 2021. μVulDeePecker: A Deep Learning-Based System for Multiclass Vulnerability Detection . *IEEE Transactions on Dependable and Secure Computing* 18, 05 (Sept. 2021), 2224–2236. doi:10.1109/TDSC.2019.2942930