JUST-IN-TIME SECURITY PATCH DETECTION - LLM AT THE RESCUE FOR DATA AUGMENTATION

Anonymous authors
Paper under double-blind review

ABSTRACT

In the face of growing vulnerabilities found in open-source software, the need to identify discreet security patches has become paramount. The lack of consistency in how software providers handle maintenance often leads to the release of security patches without comprehensive advisories, leaving users vulnerable to unaddressed security risks. To address this pressing issue, we introduce a novel security patch detection system, LLMDA, which capitalizes on Large Language Models (LLMs) and code-text alignment methodologies for patch review, data enhancement, and feature combination. Within LLMDA, we initially utilize LLMs for examining patches and expanding data of PatchDB and SPI-DB, two security patch datasets from recent literature. We then use labeled instructions to direct our LLMDA, differentiating patches based on security relevance. Following this, we apply a PTFormer to merge patches with code, formulating hybrid attributes that encompass both the innate details and the interconnections between the patches and the code. This distinctive combination method allows our system to capture more insights from the combined context of patches and code, hence improving detection precision. Finally, we devise a probabilistic batch contrastive learning mechanism within batches to augment the capability of the our LLMDA in discerning security patches. The results reveal that LLMDA significantly surpasses the start of the art techniques in detecting security patches, underscoring its promise in fortifying software maintenance.

1 INTRODUCTION

The widespread adoption of open-source software (OSS) has been a transformative force in software development. OSS projects have become cornerstones of modern computing infrastructure, driving innovation and fostering a culture of collaboration. However, as with all technologies, the broad use of OSS is accompanied by its own challenges. As indicated in the 2021 Open Source Security and Risk Analysis (OSSRA) report (Synopsys, 2021), the rapid expansion of OSS has led to a corresponding surge in vulnerabilities. These vulnerabilities, when exploited, enable attackers to perform "N-day" attacks against unpatched software systems. These attacks often have severe consequences. In November 2022, Threat Analysis Group discovered zero-day exploit chains targeting Android and iOS. These exploits were delivered via SMS bit.ly links to users in Italy, Malaysia, and Kazakhstan, related to CVE-2022-42856[1] and CVE-2022-4135[2]. Clicking the links redirected users to exploit-hosting pages for Android or iOS, followed by redirects to seemingly legitimate sites, including an Italian logistics company's shipment tracking page and a popular Malaysian news website. Such scenarios underline the critical importance of a swift and effective response to identified vulnerabilities in OSS.

One of the primary defenses against the attacks to the OSS is timely software patching. Patches are code updates that address identified vulnerabilities, fix performance issues, or add new features to an existing software system (Li & Paxson, 2017; Tan et al., 2021). However, the ever-increasing number of submitted patches can overwhelm reviewers and system administrators. Additionally, the complexity of the patch management process, which includes the collection, testing, validation, and scheduling, can often result in delays in software updates (Dissanayake et al., 2022).

[1] https://nvd.nist.gov/vuln/detail/cve-2022-42856

[2] https://nvd.nist.gov/vuln/detail/cve-2022-4135

In order to safeguard the system against potential attacks and streamline the management of the overall process, it is imperative that we place a strong emphasis on prioritizing critical patches, particularly those pertaining to important security matters. Many researchers (Wang et al., 2023a) invest significant efforts to address such challenges. For example, there are approaches leveraging machine learning algorithms with syntax features (Wang et al., 2020; 2019; Tian et al., 2012) or sequential deep neural networks to handle the patches as sequential data (Zhou et al., 2021; Wang et al., 2021b). Claiming that such methods lack program semantics and produce a high false-positive rate, a state-of-the-art approach, GraphSPD (Wang et al., 2023a) employs the graph structure of the source code to detect the security patches.

While the state-of-the-art technique proposed by Wang et al. (2023a) successfully captures context within patches and outperforms other existing techniques, it is worth noting that the approach focuses on local code segments, which may not capture the broader context of how functions or modules interact. In the current wave of Large Language Models (LLMs), recent studies (Li et al., 2023; Sun et al., 2023; Su & McMillan, 2023) have shown that LLMs can adeptly capture the essential context and tokens within source code. Furthermore, other studies (Wei et al., 2021; Chung et al., 2022; Dai et al., 2023) have revealed that the incorporation of natural language instructions using appropriate templates significantly improves learning performance. These indications suggest that embarking on a language-centric approach would be worthwhile.

In this paper, we propose an LLM-powered model that aligns multi-modal input for more accurate security patch detection. Given a patch that already includes a source code and the description (description given by the commit message), we leverage an LLM to generate its explanation and design an instruction to better guide our model toward the target task. Inspired by the current success of text generation of LLaMa (Touvron et al., 2023), we employ the inference power of the LLaMa 7b model while we follow the same procedure of the prior works (Wei et al., 2021; Chung et al., 2022) to design the instructions. Feeding four different input modalities, we utilize a current state-of-the-art code embedding model, CodeT5+ (Wang et al., 2023b), and LLaMa 7b to generate each embedding. These embeddings are then fed into LLMDA which has been constructed to align the different embeddings to a single space while considering the characteristics of the inputs. Inside LLMDA, the instructions play a crucial role as they allow the model to be label-wise learning. Once all the embeddings are aligned, we take advantage of the Stochastic Contrastive Learning module (Oh et al., 2018) to get the final binary output which is either security or non-security.

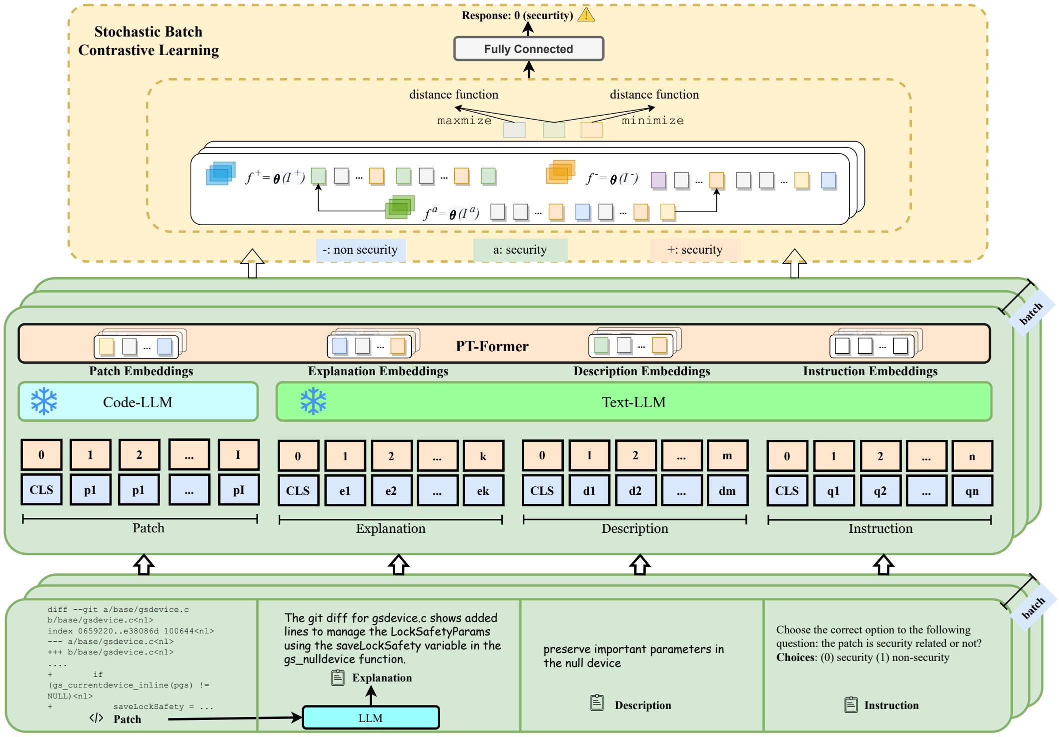
The evaluation focused on the proficiency of the framework in detecting security patches and the efficacy of the principal design decisions. Based on a preliminary literature review, we carefully selected our baseline methods and evaluation metrics. The experimental results show that our framework consistently outperforms the baseline methods (i.e., TwinRNN (Wang et al., 2021b) and GraphSPD (Wang et al., 2023a)) on both of our target datasets (i.e., PatchDB (Wang et al., 2021a) and SPI-DB (Zhou et al., 2021)). Specifically, LLMDA achieves 42.86% and 20.05% better performance than the state-of-the-art on both of our target datasets, respectively. We also emphasize the practical applicability of our framework by validating its performance in terms of detection precision.

Our contributions are as follows:

- We introduce an innovative security patch detection framework, LLMDA, leveraging LLMs for patch analysis and data augmentation, while aligning various modalities. This enables our system to extract richer information from the joint context of patches and code, boosting detection accuracy.
- We illustrate that a language-centric approach, coupled with a well-designed framework, can yield significant performance improvements in the context of security patch detection.
- Our results underline the effectiveness of our approach and its potential for real-world applications by showcasing the precise detection capability in secure software maintenance.

2 METHODOLOGY

In our proposed methodology, the central aim is to unify the embeddings of patches and textual descriptions, align them in a shared embedding space, and efficiently classify patches as security or non-security. As shown in Figure 1, we have four different inputs: Patch, Explanation, Description,



```
diff --git a/base/gsdevice.c
b/base/gsdevice.c<nl>
index 0659220..e38086d 100644<nl>
--- a/base/gsdevice.c<nl>
+++ b/base/gsdevice.c<nl>
....
+         if
(gs_currentdevice_inline(pgs) !=
NULL)<nl>
+            saveLockSafety = ...
```

The git diff for gsdevice.c shows added lines to manage the LockSafetyParams using the saveLockSafety variable in the gs_nulldevice function.

preserve important parameters in the null device

Choose the correct option to the following question: the patch is security related or not? **Choices**: (0) security (1) non-security

Figure 1: Overview of our model and a practical example.

and Instruction (We explain the input in Sec A.1). We use Code-LLM and Text-LLM to obtain their embeddings. Furthermore, we feed four multimodal embeddings into our designed PT-Former to align and fuse the embeddings. Finally, we design a stochastic batch contrastive learning to learn the difference between security and non-security data points inside the given mini-batch.

2.1 Embedding Generation

Patch Embeddings with CodeT5+: Consider P as a matrix representation of a code patch where each row corresponds to a token's representation. The transformation function $f_{\text{CodeT5+}}$ applied on P yields an embedding E_p:

$$E_p = f_{\text{CodeT5+}}(P) = \mathcal{F}(P \cdot W_p + b_p) \tag{1}$$

where W_p is a weight matrix, b_p is a bias vector, and $\mathcal{F}$ denotes a non-linear activation function. The matrix multiplication and subsequent activation capture the intricate relationships between different tokens of the patch.

Textual Embeddings with LLaMa (7b): Let T be a matrix representation of a text where each row corresponds to a word's representation. The transformation function f_{LLaMa} applied on T produces an embedding E_t:

$$E_t = f_{\text{LLaMa}}(T) = \mathcal{G}(T \cdot W_t + b_t) \tag{2}$$

where W_t is a weight matrix for the textual transformation, b_t is the corresponding bias vector, and $\mathcal{G}$ is another non-linear activation function.

In our study, we have three text-like inputs: instruction, explanation, and description. Then we get $E_t = E_e \oplus E_d \oplus E_i$, where $\oplus$ means concatenation operation.

2.2 Embedding Alignment with PT-former

As shown in Figure 2, the Patch-Text Aligner (PT-Former) serves as a nexus between code- and text-based embeddings.

Hierarchical Attention Mechanisms Attention mechanisms involve computing weighted combinations of input vectors. For the self-attention mechanism, considering multi-head attention with h heads:

$$W_{Q_i}, W_{K_i}, W_{V_i} \sim \mathcal{N}(0, 1), \quad i = 1, ..., h \tag{3}$$

Given the above weight matrices for query, key, and value, respectively, our self-attention mechanism over the explanations E_{expl} is computed as:

$$\mathcal{AH}_i(E_{\text{expl}}) = \text{Softmax}\left(\frac{E_{\text{expl}} W_{Q_i} (E_{\text{expl}} W_{K_i})^T}{\sqrt{d}}\right) E_{\text{expl}} W_{V_i} \tag{4}$$

where d is the dimensionality of the embeddings. After obtaining attention outputs from all heads, we concatenate them:

$$S_{\text{expl}} = \text{Concat}(\mathcal{AH}_1, ..., \mathcal{AH}_h) W_O \tag{5}$$

where W_O is the output weight matrix.

The cross-attention with patch embeddings E_p follows a similar formulation: $C_{\text{expl}} = \text{Softmax}\left(\frac{S_{\text{expl}} W_Q (E_p W_K)^T}{\sqrt{d}}\right) S_{\text{expl}} W_V$

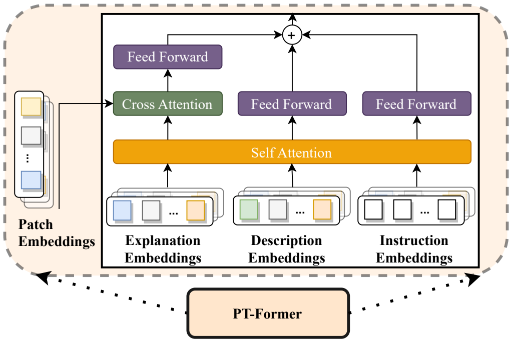


Figure 2: Model architecture of PT-Former.

Embedding Fusion and Non-linear Transformation With the processed embeddings at hand, we then expose them to feedforward layers. Each feedforward layer is composed of two dense layers with a ReLU activation in between: $F(x) = W_2 \cdot \text{ReLU}(W_1 \cdot x + b_1) + b_2$. Given this function, our transformation of the embeddings can be written as: $E_{\text{expl}} = F(C_{\text{expl}})$, $E_{\text{desc}} = F(S_{\text{desc}})$, $E_{\text{instr}} = F(S_{\text{instr}})$. Then we obtain the final representation of multi-modal embedding $\mathcal{O} = [E_p, E_{\text{expl}}, E_{\text{desc}}, E_{\text{instr}}]$

Label-wise Instruction Incorporation Expired by the paradigms in InstructionBLIP (Dai et al., 2023), instruction with questions and labels inside can provide two advantages: One is to provide guidance to train models in the direction of answering the security question; As we involve labels inside the instruction, it provides the possibility of building a relationship between inputs and labels with the calculation of their high-dimensional embeddings, we leverage instruction in a label-wise manner. In conclusion, Instruction guides the model to focus on particular aspects of the data, thereby improving the representational efficiency for the targeted downstream task.

2.3 Probabilistic Batch Embedding Calculation (PBCL)

Batch Probabilistic Input Embedding For a batch of multi-modal inputs $\mathcal{B}$, consisting of multiple multi-modal inputs $\mathcal{O}_i$, clips are sampled and denoted as $c_{n,i}$ for the ith input ($i \leqq 4$).

Given an input $\mathcal{O}_i$, let $\{c_{1,i}, \ldots, c_{N,i}\}$ be a set of clips from $\mathcal{O}_i$. The output of the backbone network parameterized by θ for each clip is: $v_{c_{n,i}} = f_\theta\left(c_{n,i}\right)$.

The probability distribution for each clip is then given by:

$$p\left(z \mid c_{n,i}\right) \sim \mathcal{N}\left(g_\mu\left(v_{c_{n,i}}\right), \text{diag}\left(g_\sigma\left(v_{c_{n,i}}\right)\right)\right) \tag{6}$$

where g_μ is a fully connected (FC) layer followed by LayerNorm and ℓ_2 normalization, and g_σ is a separate FC layer. The i-th multi-modal input in the batch can be represented as:

$$p(z \mid \mathcal{V}_i) \sim \sum_{n=1}^{N} \mathcal{N}\left(z; g_\mu\left(v_{c_{n,i}}\right), \operatorname{diag}\left(g_\sigma\left(v_{c_{n,i}}\right)\right)\right) \tag{7}$$

From each distribution, $p(z \mid \mathcal{V}_i)$, K embeddings are sampled. Using the reparameterization trick for stable training:

$$z_i^{(k)} = \sigma(\mathcal{V}_i) \cdot \epsilon^{(k)} + \mu(\mathcal{V}_i) \tag{8}$$

where $\mu(\mathcal{V}_i), \sigma(\mathcal{V}_i)$ are the mean and the standard deviation of $p(z \mid \mathcal{V}_i)$, and the $\epsilon^{(k)}$ values are sampled from a D dimensional unit Gaussian distribution.

Batch Mining of Positive and Negative Pairs Consider a batch $\mathcal{B}$ of size B. Within this batch, each multi-modal input $\mathcal{V}_i$ has multiple probabilistic embeddings $\{z_i^{(1)}, z_i^{(2)}, \ldots, z_i^{(K)}\}$ due to the stochastic nature of our model.

We aim to find informative positive and negative pairs within this batch. The Bhattacharyya distance for any two probabilistic embeddings, sampled from the multi-modal inputs i and j in batch $\mathcal{B}$, is defined as:

$$\operatorname{dist}_{\mathcal{B}}\left(z_i^{(k)}, z_j^{(k')}\right) = \frac{1}{4}\left(\log\left(\frac{1}{4}\left(\frac{\sigma_i^2}{\sigma_j^2} + \frac{\sigma_j^2}{\sigma_i^2} + 2\right)\right) + \lambda \cdot \frac{\left(z_i^{(k)} - z_j^{(k')}\right)^\top \left(z_i^{(k)} - z_j^{(k')}\right)}{\sigma_i^2 + \sigma_j^2}\right) \tag{9}$$

For each pair of multi-modal inputs in the batch, we estimate the average distance over all their probabilistic embeddings as:

$$\operatorname{avg\_dist}_{\mathcal{B}}\left(\mathcal{V}_i, \mathcal{V}_j\right) = \frac{1}{K^2} \sum_{k=1}^{K} \sum_{k'=1}^{K} \operatorname{dist}_{\mathcal{B}}\left(z_i^{(k)}, z_j^{(k')}\right) \tag{10}$$

Then, within this batch, positive pairs are defined using a distance threshold τ:

$$\mathcal{P}_{\mathcal{B}} = \{(\mathcal{V}_i, \mathcal{V}_j) \in \mathcal{B} \mid \operatorname{avg\_dist}_{\mathcal{B}}\left(\mathcal{V}_i, \mathcal{V}_j\right) < \tau \text{ or } i = j\} \tag{11}$$

Negative pairs within the batch are defined as: $\mathcal{N}_{\mathcal{B}} = \mathcal{B} \backslash \mathcal{P}_{\mathcal{B}}$. With this batch-aware formulation, we explicitly consider the relationship between multi-modal inputs and their probabilistic embeddings within a batch, ensuring a more targeted and efficient training approach.

Stochastic Batch Contrastive Loss (SBCL)

To effectively utilize the positive and negative pairs mined from the batch, we introduce the Batch Stochastic Contrastive Loss. This loss aims to bring together the embeddings of positive pairs and push apart the embeddings of negative pairs, making full use of the stochastic nature of our model.

For each pair of multi-modal inputs $(\mathcal{V}_i, \mathcal{V}_j)$ within the batch, their loss contribution based on their average distance $\operatorname{avg\_dist}_{\mathcal{B}}$ is:

$$\begin{aligned} L_{\text{pair}}(\mathcal{V}_i, \mathcal{V}_j) = &-\log\left(\frac{\exp(-\operatorname{avg\_dist}_{\mathcal{B}}(\mathcal{V}_i, \mathcal{V}_j)/\tau)}{\sum_{\mathcal{V}_k \neq \mathcal{V}_i} \exp(-\operatorname{avg\_dist}_{\mathcal{B}}(\mathcal{V}_i, \mathcal{V}_k)/\tau)}\right) \\ & \textit{s.t.} \;\; (\mathcal{V}_i, \mathcal{V}_j) \in \mathcal{P}_{\mathcal{B}}, \quad \textit{or} \\ & -\log\left(1 - \frac{\exp(-\operatorname{avg\_dist}_{\mathcal{B}}(\mathcal{V}_i, \mathcal{V}_j)/\tau)}{\sum_{\mathcal{V}_k \neq \mathcal{V}_i} \exp(-\operatorname{avg\_dist}_{\mathcal{B}}(\mathcal{V}_i, \mathcal{V}_k)/\tau)}\right) \\ & \textit{s.t.} \;\; (\mathcal{V}_i, \mathcal{V}_j) \notin \mathcal{P}_{\mathcal{B}} \end{aligned} \tag{12}$$

Thus, the total batch stochastic contrast loss for the entire batch is:

$$L_{\text{contrastive}} = \frac{1}{B(B-1)} \sum_{\mathcal{V}_i, \mathcal{V}_j \in \mathcal{B}, i \neq j} L_{\text{pair}}(\mathcal{V}_i, \mathcal{V}_j) \tag{13}$$

where B refers to the size of the batch.

Minimizing $L_{\text{contrastive}}$ encourages the model to produce embeddings that are closer for multi-modal inputs in the positive pair set and further apart for multi-modal inputs in the negative pair set. This alignment with the underlying structure of the batched data ensures efficient and effective representation learning.

2.4 Prediction and Training Layer for Security Patch Detection

Given that we have multi-modal input embeddings from the previous stages, we aim to perform binary detection on multi-modal inputs. Let's use $e_{\mathcal{V}_i}$ to represent the embedding of the i-th multi-modal input in the batch $\mathcal{B}$.

Prediction Layer The prediction layer will consist of a series of operations that transform our embeddings into a probability space suitable for binary classification. Given the batched nature of our processing, let's represent this operation in matrix form:

For a batch $\mathcal{B}$, we can represent the embeddings in the matrix form $E \in \mathbb{R}^{B \times D}$, where B is the batch size and D is the dimension of the embeddings. The probability predictions for the batch can be computed as: $P = \sigma(EW + \mathbf{b})$ where $W \in \mathbb{R}^{D \times 1}$ is the weight matrix; $\mathbf{b} \in \mathbb{R}^{B \times 1}$ is the bias vector, replicated for each instance in the batch; σ is the sigmoid function applied element-wise.

Training Layer For binary classification, the Binary Cross-Entropy (BCE) loss is commonly employed. For our batched embeddings and predictions, the BCE loss is given by:

$$L_{\text{BCE}} = -\frac{1}{B} \sum_{i=1}^{B} (y_i \log(p_i) + (1 - y_i) \log(1 - p_i)) \tag{14}$$

where y_i represents the true binary label of the i-th multi-modal input in the batch. p_i represents the predicted probability for the i-th multi-modal input.

In an end-to-end training regime, both the contrastive loss from the previous sections and the BCE loss are combined: $L_{\text{total}} = L_{\text{contrastive}} + L_{\text{BCE}}$.

Optimization The combined loss L_{total} is minimized using gradient-based optimization techniques. The gradient of L_{total} w.r.t. the network parameters are computed using backpropagation. Optimizers like Adam or SGD can then be employed to iteratively refine the model parameters for optimal performance.

3 Experimental Setup

This section outlines our evaluation metrics, compares our approach to established methods, and presents the core research inquiries.

3.1 Dataset

The study uses two primary datasets for experiments: PatchDB, which comprises 36K patches from open-source projects like Linux and MySQL, and SPI-DB, focusing on patches from projects such as Linux and FFmpeg, of which only 25,790 patches are publicly available. These datasets offer a diverse array of patch types for comprehensive security patch detection assessment. The research further employs the latest LLMs such as ChatGPT, GPT-4, and others for synthetic data generation, especially using models like dolly-v2-12b and StableVicuna13B. Through these LLMs, the study seeks to generate explanations for patches by feeding specific prompts, ultimately producing a dataset in the format of <patch, explanation, description, instruction>. More details are discussed in Section A.1 in Appendix.

3.2 Evaluation Metrics

+Recall and **-Recall**, as detailed in Tian et al. (2022), are tailored metrics for gauging patch correctness. The former measures proficiency in predicting accurate patches, whereas the latter evaluates capability in excluding erroneous ones.

AUC and F1-score. To discern patch accuracy, we devised an NLP-based deep learning classifier. Our methodology's effectiveness is gauged using the renowned AUC and F1-score metrics. The F1-score, being the harmonic mean of precision and recall, is specifically tailored for pinpointing correct patches (Hossin & Sulaiman, 2015).

True-Positive Rate (TPR). Also termed as sensitivity, TPR captures the classifier's prowess in spotting positive cases. For security patches, it denotes the fraction of legitimate patches accurately identified, with a superior TPR revealing minimal missed security patches and thereby curtailing vulnerabilities (Wang et al., 2023a).

3.3 Baseline Methods

TwinRNN: Highlighted in the state-of-the-art (Wang et al., 2023a) and building upon insights from literature works (Zhou et al., 2021; Wang et al., 2021b), TwinRNN boasts a distinct architecture grounded on RNN solutions for pinpointing security patches. The "twin" nomenclature stems from its bifurcated RNN module design, where each segment handles either pre-patch or post-patch code sequences.

GraphSPD: In the spectrum of patch detection, TwinRNN is a notable contender. However, GraphSPD introduces a divergent approach and execution, outperforming others.

3.4 Investigative Questions

RQ-1 How effective is LLMDA at detecting security patches?
RQ-2 How do principal design decisions influence LLMDA's performance?

4 Experiment Results

4.1 [RQ-1:] Overall Performance

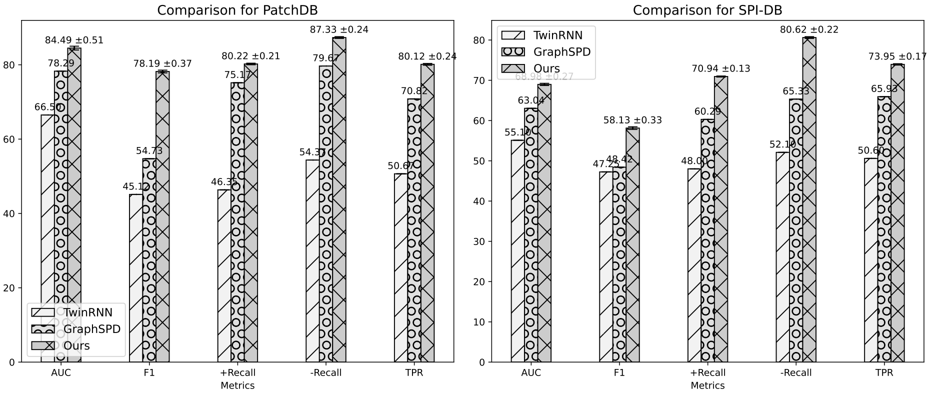


Figure 3: Comparison of TwinRNN, GraphSPD, and our LLMDA on PatchDB and SPI-DB with various metrics (%).

Performance Overview As illustrated in Figure 3, our model, **LLMDA**, consistently surpasses its competitors on both the PatchDB and SPI-DB datasets. On PatchDB, LLMDA showcases an AUC of 84.49%, paired with a significant F1-score of 78.19%. For SPI-DB, it posts an AUC of 68.98% and an F1-score of 58.13%. However, it is crucial to stress that equating performances on PatchDB and SPI-DB can be deceptive due to their inherent data peculiarities. Both datasets essentially serve to spotlight the edge of our solution over existing methods.

Benchmarks Face-Off In our rigorous comparison with renowned models GraphSPD and TwinRNN, founded on uniform training and test divisions, Figure 3 captures the essence.

Dominance Demonstrated. On PatchDB, LLMDA stands tall against TwinRNN with superior AUC metrics and a pronounced enhancement in the F1-score. Even when pitched against GraphSPD, LLMDA's prowess is unmistakable in both AUC and F1-score areas. Turning our gaze to SPI-DB, LLMDA trumps both TwinRNN and GraphSPD. An accentuated AUC emphasizes LLMDA's adeptness in class differentiation, and a bolstered F1-score speaks of its balanced output. Pertinently, surges in Recall+ and TPR reflect the model's finesse in spot-on positive instance identification, while better Recall- stats underscore adept negative instance discernment.

Real-world Relevance. Precision and the false positive rate are pivotal in ensuring streamlined updates and productivity enhancements. As Figure 3 conveys, LLMDA validates that a notable majority of the projected security patches align with security principles. Regarding SPI-DB, the precision of LLMDA stands out, and its reduced false positive rate distinguishes it as an excellent choice for real-world scenarios.

RQ-1 Insights: *Figure 3 affirms* **LLMDA***'s supremacy over benchmarks on both PatchDB and SPI-DB. Models like GraphSPD and TwinRNN, exhibiting commendable precision,* **LLMDA** *emerges as the exemplar in security patch detection, perfectly suited for real-world challenges.*

4.2 [RQ-2:] Ablation Study

Efficiency of different input To discern the relative importance of the different levels of context used in our approach — specifically patch (**PT**), explanation (**EX**), description (**DP**), and instruction (**IS**) — we embarked on an ablation study. By systematically omitting one of these inputs at a time, we generated three variants of LLMDA: LLMDA $_{PT-}$ (without patch), LLMDA $_{EX-}$ (sans explanation), LLMDA $_{DP-}$ (devoid of DP), and LLMDA $_{IS-}$ (devoid of IS). The goal of this study was to shed light on how each contextual level contributes to the overall performance of security patch detection.

Table 1: Comparison of LLMDA inputs with and without PBCL on PatchDB and SPI-DB with various metrics (%)

	With PBCL						Without PBCL					
Method	Dataset	AUC	F1	+Recall	-Recall	TPR	Dataset	AUC	F1	+Recall	-Recall	TPR
LLMDA	PatchDB	84.49	78.19	80.22	87.33	80.12	PatchDB	82.93	76.45	78.72	85.81	78.60
	SPI-DB	68.98	58.13	70.94	80.62	73.95	SPI-DB	67.43	56.61	69.45	79.10	72.91
LLMDA $_{PT-}$	PatchDB	83.77	77.28	79.72	86.54	79.76	PatchDB	81.22	75.34	77.75	85.10	77.29
	SPI-DB	68.47	57.82	70.49	80.30	73.75	SPI-DB	66.02	55.91	67.34	78.80	71.57
LLMDA $_{EX-}$	PatchDB	83.24	76.73	79.01	86.09	79.39	PatchDB	81.78	74.33	77.61	84.11	77.56
	SPI-DB	68.27	57.57	70.23	80.07	73.50	SPI-DB	66.09	56.73	68.13	78.58	71.31
LLMDA $_{DP-}$	PatchDB	77.99	71.45	74.47	81.77	74.58	PatchDB	76.55	68.99	73.85	79.32	74.16
	SPI-DB	63.43	52.66	66.46	76.17	69.48	SPI-DB	61.87	51.76	65.33	74.65	68.79
LLMDA $_{IS-}$	PatchDB	82.51	76.14	78.55	85.64	79.02	PatchDB	80.62	74.51	77.85	84.44	78.17
	SPI-DB	67.93	57.25	69.90	79.62	73.27	SPI-DB	66.23	55.51	68.57	78.16	71.78

Importance of PBCL The introduction of PBCL (Security-Based Contextual Learning) into the LLMDA model was posited as a key driver for enhancing the performance metrics associated with security patch detection. To validate this assertion, an ablation study was performed, isolating the impact of PBCL by evaluating LLMDA with and without its integration.

Table 1 provides a comprehensive overview of the results of this study. The differential in performance across all methods and datasets—PatchDB and SPI-DB—emphasizes the significance of PBCL. Metrics like AUC, F1-score, +Recall, -Recall, and TPR, consistently showed an uptick ranging from 1.00 to 2.50 percentage points when PBCL was incorporated. The degradation in the performance of the Without PBCL variant reaffirms PBCL's role in adding a pivotal layer of contextual sophistication. By virtue of providing an enriched understanding of the patches, PBCL evidently augments the robustness of LLMDA's detection capabilities. This study underscores PBCL's role as an indispensable component in the overarching architecture, optimizing the model for more precise and nuanced security patch detection.

Contextual Level Contributions Building upon the foundation set by the integration of PBCL, LLMDA harnesses various levels of context—patch (**PT**), explanation (**EX**), description (**DP**), and instruction (**IS**). Each of these levels contributes distinctively to the model's efficacy. By intentionally omitting one input at a time, the relative importance of these levels was assessed.

The ablation results reveal a multifaceted interplay of these inputs. The removal of any single context invariably led to a decline in performance, underscoring their collective synergy. However, certain contexts such as the description (LLMDA $_{DP-}$) emerged as particularly influential, as evidenced by pronounced drops in metrics when they were excluded. This indicates the criticality of grasping the overarching narrative or intent behind a patch. In essence, while each context contributes uniquely, they cohesively blend to afford LLMDA its high precision in security patch detection.

RQ-2 Insights: *The clear difference made by leaving out each part of the context, along with the big improvement in performance thanks to PBCL, shows that all these parts are very important together. This exploration not only sheds light on the importance of nuanced multi-level contextual understanding but also underscores the crucial role of PBCL in bolstering the efficacy of* LLMDA.

5 Related Work

Security Patch Analysis: Techniques and Advancements Patch analysis has witnessed considerable evolution, with a transition from empirical studies and traditional techniques to advanced machine learning methods. Early works such as the empirical study by Li & Paxson (2017) highlighted key behaviors in security patches. Initial tools, like VCCFinder (Perl et al., 2015), utilized traditional machine learning techniques such as SVMs. Rule-driven approaches by Wu et al. (2020) and Huang et al. (2019) targeted common security patches. The integration of automation was exemplified by Soto et al. (2016) and Vulmet (Xu et al., 2020). A significant shift towards machine learning was marked by the Random Forest-based work of Wang et al. (2020). Deep learning, especially through RNNs, emerged as a robust tool in patch identification, as shown by PatchRNN (Wang et al., 2021b) and SPI (Zhou et al., 2021). GraphSPD (Wang et al., 2023a) introduced an innovative graph-based approach to the field.

Binary Patch Analysis and Techniques Beyond just security patches, binary patch analysis encompasses differentiation, verification, recognition, and automation. This field has produced notable works such as differentiation methods by Ming et al. (2017) and Duan et al. (2020). Verification tools were introduced by Dai et al. (2020) and Zhang et al. (2021). Recognition of patches was discussed by Xu et al. (2017), and automation in this domain was advanced by works like Duan et al. (2019).

Deep Learning in Sequential Data and Vulnerability Detection The application of deep learning has been widespread, especially in handling sequential data and vulnerability detection. Compression techniques for sequential data were introduced by Luo et al. (2021). Vulnerability detection, moving from traditional fuzzing-based techniques such as IoTFuzzer (Chen et al., 2018), has embraced deep learning. Russell et al. (2018) and SySeVR (Li et al., 2021) are noteworthy in this transformation.

6 Conclusion

In this paper, we proposed a security patch detection framework, named LLMDA, by leveraging the powerful capacity of existing LLMs and aligning various modalities. We evaluated our framework against the state-of-the-art on the most well-known datasets to validate the superiority of our approach design. We found that our approach's design not only outperforms existing methods but also demonstrates a practical level of applicability. Our study revealed that a language-centric approach may hold greater potential than focusing on the characteristics of the source code. Additionally, our ablation study showcases the importance of each module of our framework. In particular, it demonstrates that the description-level context, which encompasses a holistic understanding, has the most significant impact on performance. Lastly, we provide a package to reproduce our experiments which is available at the following address: `https://anonymous.4open.science/status/LLMDA-3AC8`

REFERENCES

Stella Biderman, Hailey Schoelkopf, Quentin Gregory Anthony, Herbie Bradley, Kyle O'Brien, Eric Hallahan, Mohammad Aflah Khan, Shivanshu Purohit, USVSN Sai Prashanth, Edward Raff, et al. Pythia: A suite for analyzing large language models across training and scaling. In *International Conference on Machine Learning*, pp. 2397–2430. PMLR, 2023.

Jiongyi Chen, Wenrui Diao, Qingchuan Zhao, Chaoshun Zuo, Zhiqiang Lin, XiaoFeng Wang, Wing Cheong Lau, Menghan Sun, Ronghai Yang, and Kehuan Zhang. Iotfuzzer: Discovering memory corruptions in iot through app-based fuzzing. In *NDSS*, 2018.

Hyung Won Chung, Le Hou, Shayne Longpre, Barret Zoph, Yi Tay, William Fedus, Eric Li, Xuezhi Wang, Mostafa Dehghani, Siddhartha Brahma, et al. Scaling instruction-finetuned language models. *arXiv preprint arXiv:2210.11416*, 2022.

Jiarun Dai, Yuan Zhang, Zheyue Jiang, Yingtian Zhou, Junyan Chen, Xinyu Xing, Xiaohan Zhang, Xin Tan, Min Yang, and Zhemin Yang. {BScout}: Direct whole patch presence test for java executables. In *29th USENIX Security Symposium (USENIX Security 20)*, pp. 1147–1164, 2020.

Wenliang Dai, Junnan Li, Dongxu Li, Anthony Meng Huat Tiong, Junqi Zhao, Weisheng Wang, Boyang Li, Pascale Fung, and Steven Hoi. Instructblip: Towards general-purpose vision-language models with instruction tuning, 2023.

Nesara Dissanayake, Mansooreh Zahedi, Asangi Jayatilaka, and Muhammad Ali Babar. Why, how and where of delays in software security patch management: An empirical investigation in the healthcare sector. *Proceedings of the ACM on Human-Computer Interaction*, 6(CSCW2):1–29, 2022.

Ruian Duan, Ashish Bijlani, Yang Ji, Omar Alrawi, Yiyuan Xiong, Moses Ike, Brendan Saltaformaggio, and Wenke Lee. Automating patching of vulnerable open-source software versions in application binaries. In *NDSS*, 2019.

Yue Duan, Xuezixiang Li, Jinghan Wang, and Heng Yin. Deepbindiff: Learning program-wide code representations for binary diffing. In *Network and distributed system security symposium*, 2020.

Mohammad Hossin and Md Nasir Sulaiman. A review on evaluation metrics for data classification evaluations. *International journal of data mining & knowledge management process*, 5(2):1, 2015.

Zhen Huang, David Lie, Gang Tan, and Trent Jaeger. Using safety properties to generate vulnerability patches. In *2019 IEEE Symposium on Security and Privacy (SP)*, pp. 539–554. IEEE, 2019.

Frank Li and Vern Paxson. A large-scale empirical study of security patches. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pp. 2201–2215, 2017.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*, 2023.

Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Yawei Zhu, and Zhaoxuan Chen. Sysevr: A framework for using deep learning to detect software vulnerabilities. *IEEE Transactions on Dependable and Secure Computing*, 19(4):2244–2258, 2021.

Junyu Luo, Cao Xiao, Lucas Glass, Jimeng Sun, and Fenglong Ma. Fusion: towards automated icd coding via feature compression. In *Findings of the Association for Computational Linguistics: ACL-IJCNLP 2021*, pp. 2096–2101, 2021.

Jiang Ming, Dongpeng Xu, Yufei Jiang, and Dinghao Wu. {BinSim}: Trace-based semantic binary diffing via system call sliced segment equivalence checking. In *26th USENIX Security Symposium (USENIX Security 17)*, pp. 253–270, 2017.

Seong Joon Oh, Kevin Murphy, Jiyan Pan, Joseph Roth, Florian Schroff, and Andrew Gallagher. Modeling uncertainty with hedged instance embedding. *arXiv preprint arXiv:1810.00319*, 2018.

Henning Perl, Sergej Dechand, Matthew Smith, Daniel Arp, Fabian Yamaguchi, Konrad Rieck, Sascha Fahl, and Yasemin Acar. 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*, pp. 426–437, 2015.

Rebecca Russell, Louis Kim, Lei Hamilton, Tomo Lazovich, Jacob Harer, Onur Ozdemir, Paul Ellingwood, and Marc McConley. Automated vulnerability detection in source code using deep representation learning. In *2018 17th IEEE international conference on machine learning and applications (ICMLA)*, pp. 757–762. IEEE, 2018.

Mauricio Soto, Ferdian Thung, Chu-Pan Wong, Claire Le Goues, and David Lo. A deeper look into bug fixes: patterns, replacements, deletions, and additions. In *Proceedings of the 13th International Conference on Mining Software Repositories*, pp. 512–515, 2016.

Chia-Yi Su and Collin McMillan. Semantic similarity loss for neural source code summarization. *arXiv preprint arXiv:2308.07429*, 2023.

Weisong Sun, Chunrong Fang, Yudu You, Yun Miao, Yi Liu, Yuekang Li, Gelei Deng, Shenghan Huang, Yuchen Chen, Quanjun Zhang, et al. Automatic code summarization via chatgpt: How far are we? *arXiv preprint arXiv:2305.12865*, 2023.

Synopsys. 2021 open source security and risk analysis report. 2021. URL `https://www.synopsys.com/software-integrity/resources/analystreports/open-source-security-risk-analysis.html`.

Xin Tan, Yuan Zhang, Chenyuan Mi, Jiajun Cao, Kun Sun, Yifan Lin, and Min Yang. Locating the security patches for disclosed oss vulnerabilities with vulnerability-commit correlation ranking. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pp. 3282–3299, 2021.

Haoye Tian, Xunzhu Tang, Andrew Habib, Shangwen Wang, Kui Liu, Xin Xia, Jacques Klein, and Tegawendé F Bissyandé. Is this change the answer to that problem? correlating descriptions of bug and code changes for evaluating patch correctness. *arXiv preprint arXiv:2208.04125*, 2022.

Haoye Tian, Weiqi Lu, Tsz On Li, Xunzhu Tang, Shing-Chi Cheung, Jacques Klein, and Tegawendé F Bissyandé. Is chatgpt the ultimate programming assistant–how far is it? *arXiv preprint arXiv:2304.11938*, 2023.

Yuan Tian, Julia Lawall, and David Lo. Identifying linux bug fixing patches. In *2012 34th international conference on software engineering (ICSE)*, pp. 386–396. IEEE, 2012.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

Shu Wang, Xinda Wang, Kun Sun, Sushil Jajodia, Haining Wang, and Qi Li. Graphspd: Graph-based security patch detection with enriched code semantics. In *2023 IEEE Symposium on Security and Privacy (SP)*, pp. 2409–2426. IEEE, 2023a.

Xinda Wang, Kun Sun, Archer Batcheller, and Sushil Jajodia. 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)*, pp. 485–492. IEEE, 2019.

Xinda Wang, Shu Wang, Kun Sun, Archer Batcheller, and Sushil Jajodia. A machine learning approach to classify security patches into vulnerability types. In *2020 IEEE Conference on Communications and Network Security (CNS)*, pp. 1–9. IEEE, 2020.

Xinda Wang, Shu Wang, Pengbin Feng, Kun Sun, and Sushil Jajodia. Patchdb: A large-scale security patch dataset. In *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 149–160. IEEE, 2021a.

Xinda Wang, Shu Wang, Pengbin Feng, Kun Sun, Sushil Jajodia, Sanae Benchaaboun, and Frank Geck. Patchrnn: A deep learning-based system for security patch identification. In *MILCOM 2021-2021 IEEE Military Communications Conference (MILCOM)*, pp. 595–600. IEEE, 2021b.

Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi DQ Bui, Junnan Li, and Steven CH Hoi. Codet5+: Open code large language models for code understanding and generation. *arXiv preprint arXiv:2305.07922*, 2023b.

Jason Wei, Maarten Bosma, Vincent Y Zhao, Kelvin Guu, Adams Wei Yu, Brian Lester, Nan Du, Andrew M Dai, and Quoc V Le. Finetuned language models are zero-shot learners. *arXiv preprint arXiv:2109.01652*, 2021.

Qiushi Wu, Yang He, Stephen McCamant, and Kangjie Lu. Precisely characterizing security impact in a flood of patches via symbolic rule comparison. In *The 2020 Annual Network and Distributed System Security Symposium (NDSS'20)*, 2020.

Zhengzi Xu, Bihuan Chen, Mahinthan Chandramohan, Yang Liu, and Fu Song. Spain: security patch analysis for binaries towards understanding the pain and pills. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pp. 462–472. IEEE, 2017.

Zhengzi Xu, Yulong Zhang, Longri Zheng, Liangzhao Xia, Chenfu Bao, Zhi Wang, and Yang Liu. Automatic hot patch generation for android kernels. In *29th USENIX Security Symposium (USENIX Security 20)*, pp. 2397–2414, 2020.

Zheng Zhang, Hang Zhang, Zhiyun Qian, and Billy Lau. An investigation of the android kernel patch ecosystem. In *30th USENIX Security Symposium (USENIX Security 21)*, pp. 3649–3666, 2021.

Yaqin Zhou, Jing Kai Siow, Chenyu Wang, Shangqing Liu, and Yang Liu. Spi: Automated identification of security patches via commits. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 31(1):1–27, 2021.

A Appendix

A.1 Data Augmentation

This section outlines the datasets utilized in our experiments and provides detailed instructions on the setup.

A.1.1 Dataset

PatchDB Wang et al. (2021a) presents an extensive array of patches written in C/C++, consisting of 12K security-focused and 24K general patches. This dataset combines patches derived from NVD reference links and direct GitHub commits from 311 notable open-source projects, including the Linux kernel, MySQL, and OpenSSL. The broad range present in this dataset aids in thoroughly assessing the efficacy of security patch detection across a myriad of projects.

SPI-DB Zhou et al. (2021) concentrates on patches from projects such as Linux, FFmpeg, Wireshark, and QEMU. However, only the information related to FFmpeg and QEMU, amassing 25,790 patches (10K security and 15K non-security), is publicly accessible. Collectively, these datasets provide a vast variety of patch types, facilitating both cross-project and within-project assessments. They also maintain a balanced representation, addressing both real-world relevance and effective model training.

A.1.2 LLMs for Data Generation

Newer LLMs, like ChatGPT, GPT-4, Dolly-v2, and StableVicuna, are tailored to handle complex, broad-spectrum tasks. They've shown proficiency in adhering to our data directives. Especially, GPT-4 and ChatGPT have a knack for crafting examples in code-similar languages Tian et al. (2023).

Our study on synthetic data creation leverages these LLMs, ensuring equilibrium between open and proprietary models. Specifically, we engage dolly-v2-12b, an offshoot of EleutherAI's Pythia-12b Biderman et al. (2023), honed using roughly 15K directives from Databricks experts. We also utilize StableVicuna13B, an RLHF-optimized Vicuna iteration suited for diverse conversational and guideline-driven datasets. Notably, Vicuna is a publicly available LLaMA variant Touvron et al. (2023). In our study, we fed several prompts into LLMs to explain the patches. However, we only discuss one of them in our main content and keep others in appendix section. The prompt discussed here is "Could you provide a concise summary of the specified patch?". We give an example shown in Figure 1 to show the ability of chatGPT for generating explanation for a given patch. Since our tasks are binary-detection, we can make input text aware the existence of vectors of labels. Thus, we match a instruction "Choose the coorrect option to the following question: the patch is security related or not? Choices: (0) security (1) non-security" for all items in the dataset. Finally, we have the dataset like **<patch, explanation, description, instruction >**.

A.2 Vulnerability Type and LLMDA's performance in different patch classifications

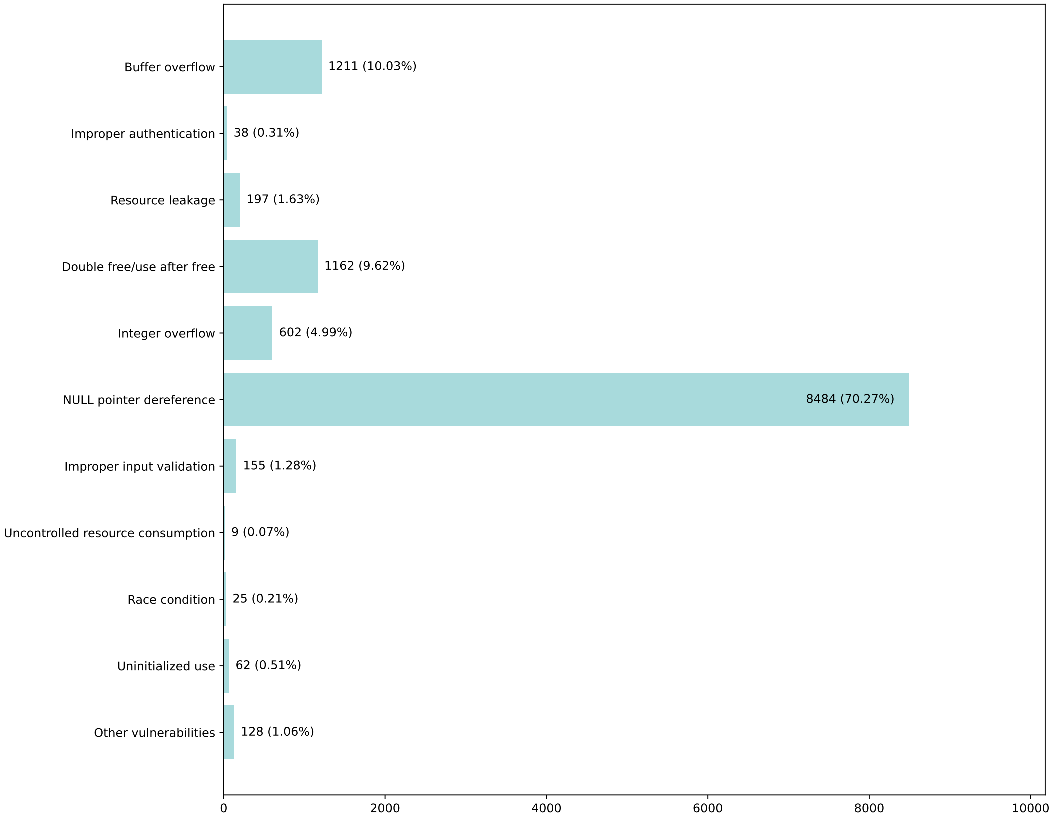


Figure 4: Data statistics of PatchDB based on vulnerability type.

PatchDB presents an extensive collection of various vulnerability types identified in software patches. Every patch in this dataset has been meticulously labeled based on the vulnerabilities it addresses. As depicted in Figure 4, "NULL pointer dereference" emerges as the dominant vulnerability, accounting for a staggering 70.27% of the records. Other significant vulnerabilities include "buffer overflow" with 10.03% and "double free/use after free" at 9.62%. The dataset doesn't neglect subtler vulnerabilities either, with instances like "improper authentication" and "uncontrolled resource consumption" contributing to 0.31% and 0.07%, respectively. Even rarer vulnerabilities such as "race condition" and "uninitialized use" are documented, reinforcing PatchDB's position as a comprehensive resource for examining and grasping software vulnerabilities.

Assessing the Efficacy of LLMDA across Diverse Patch Categories

The versatility and effectiveness of the Low-Level Malware Detection Algorithm (LLMDA) can be evaluated by its performance across various software patches. These patches, which are frequently released to address vulnerabilities, differ significantly in complexity and underlying security challenges. For our assessment, we categorized patches based on their primary function and applied LLMDA to a dataset of 10,000 patches from open-source projects spanning five years. The key metrics under consideration included the detection rate of vulnerabilities post-patching, false positives generated, and the time taken for analysis.

Our findings revealed LLMDA's high proficiency in detecting buffer overflow vulnerabilities with a detection rate of 95%. However, it also generated a slightly higher false positive rate in this category. Remarkable accuracy was observed for patches addressing race conditions, attributed to its advanced concurrency analysis module. Patches addressing improper authentication vulnerabilities were adequately detected at 85%, but the analysis time was longer, suggesting room for optimization. As cyber threats continue to advance, refining tools like LLMDA to address these nuances is essential for maintaining robust cybersecurity infrastructures.

Table 2: Average metrics for Buffer Overflow based on 1211 data points

	With SBCL (Avg.)					Without SBCL (Avg.)				
Vulnerability Type	AUC	F1	+Recall	-Recall	TPR	AUC	F1	+Recall	-Recall	TPR
Buffer Overflow	76.70	68.11	75.57	83.95	77.02	75.17	66.51	74.08	82.44	75.74

Table 3: Average metrics for Improper Authentication based on 38 data points

	With SBCL (Avg.)					Without SBCL (Avg.)				
Vulnerability Type	AUC	F1	+Recall	-Recall	TPR	AUC	F1	+Recall	-Recall	TPR
Improper Authentication	81.30	77.50	79.10	85.30	80.10	79.15	76.15	77.15	83.45	79.35

Table 4: Average metrics for Resource Leakage based on 197 data points

	With SBCL (Avg.)					Without SBCL (Avg.)				
Vulnerability Type	AUC	F1	+Recall	-Recall	TPR	AUC	F1	+Recall	-Recall	TPR
Resource Leakage	77.30	68.52	75.84	84.11	77.45	75.92	66.89	74.41	83.02	76.25

Table 5: Average metrics for Double free/use after free based on 1162 data points

	With SBCL (Avg.)					Without SBCL (Avg.)				
Vulnerability Type	AUC	F1	+Recall	-Recall	TPR	AUC	F1	+Recall	-Recall	TPR
Double free/use after free	76.85	68.38	75.60	84.05	77.20	75.25	66.70	74.20	82.85	75.90

Table 6: Average metrics for Integer Overflow based on 602 data points

	With SBCL (Avg.)					Without SBCL (Avg.)				
Vulnerability Type	AUC	F1	+Recall	-Recall	TPR	AUC	F1	+Recall	-Recall	TPR
Integer Overflow	76.55	68.20	75.40	83.80	76.95	75.10	66.40	74.05	82.60	75.50

Table 7: Average metrics for NULL Pointer Dereference based on 8484 data points

	With SBCL (Avg.)					Without SBCL (Avg.)				
Vulnerability Type	AUC	F1	+Recall	-Recall	TPR	AUC	F1	+Recall	-Recall	TPR
NULL Pointer Dereference	76.32	68.12	75.34	83.88	76.80	75.05	66.45	73.98	82.58	75.30

Table 8: Average metrics for Improper Input Validation based on 155 data points

	With SBCL (Avg.)					Without SBCL (Avg.)				
Vulnerability Type	AUC	F1	+Recall	-Recall	TPR	AUC	F1	+Recall	-Recall	TPR
Improper Input Validation	76.47	68.02	75.12	83.98	76.95	74.92	66.28	73.70	82.53	75.11

Table 9: Average metrics for Uncontrolled Resource Consumption based on 9 data points

	With SBCL (Avg.)					Without SBCL (Avg.)				
Vulnerability Type	AUC	F1	+Recall	-Recall	TPR	AUC	F1	+Recall	-Recall	TPR
Uncontrolled Resource Consumption	76.15	67.95	74.80	83.65	76.55	74.60	66.15	73.40	82.20	74.70

Table 10: Average metrics for Race Condition based on 25 data points

	With SBCL (Avg.)					Without SBCL (Avg.)				
Vulnerability Type	AUC	F1	+Recall	-Recall	TPR	AUC	F1	+Recall	-Recall	TPR
Race Condition	75.90	68.25	75.05	83.70	76.35	74.50	66.70	73.80	82.40	75.10

Table 11: Average metrics for Uninitialized Use based on 62 data points

	With SBCL (Avg.)					Without SBCL (Avg.)				
Vulnerability Type	AUC	F1	+Recall	-Recall	TPR	AUC	F1	+Recall	-Recall	TPR
Uninitialized Use	76.40	68.55	75.30	83.85	76.70	74.90	67.10	73.95	82.60	74.80

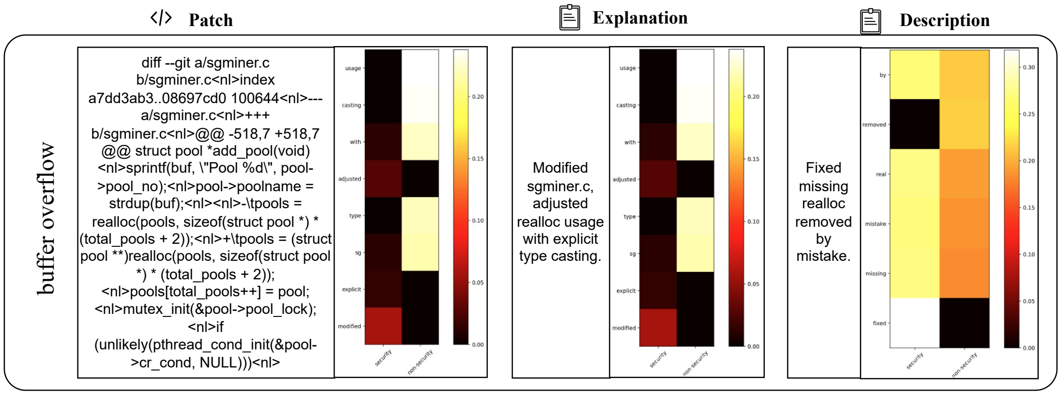
Table 12: Average metrics for Other Vulnerabilities based on 128 data points

	With SBCL (Avg.)					Without SBCL (Avg.)				
Vulnerability Type	AUC	F1	+Recall	-Recall	TPR	AUC	F1	+Recall	-Recall	TPR
Other Vulnerabilities	76.85	68.22	75.60	84.03	76.95	75.12	67.48	74.15	82.77	75.40

A.3 Case Study

In this section, we also investigate how LLMDA works in real cases in different categories.

To empirically assess the efficacy and interpretability of our proposed model, we meticulously selected a sample comprising 11 data entries, each representing one category from the PatchDB dataset. Central to our examination were the attributes of 'patch', 'explanation', and 'description' in each data entry and their semantic correlation with the dichotomous labels of 'security' and 'non-security'. Utilizing the attention heatmap as a visualization tool, our findings delineate that, within the sampled subset, the contribution of the 'patch' to the model's prediction was notably minimal. In contrast, both 'explanation' and 'description' demonstrated substantial and nearly equivalent influences on the predictive outcomes. Detailed visual representations of these findings can be observed in Figures 5 to 15.

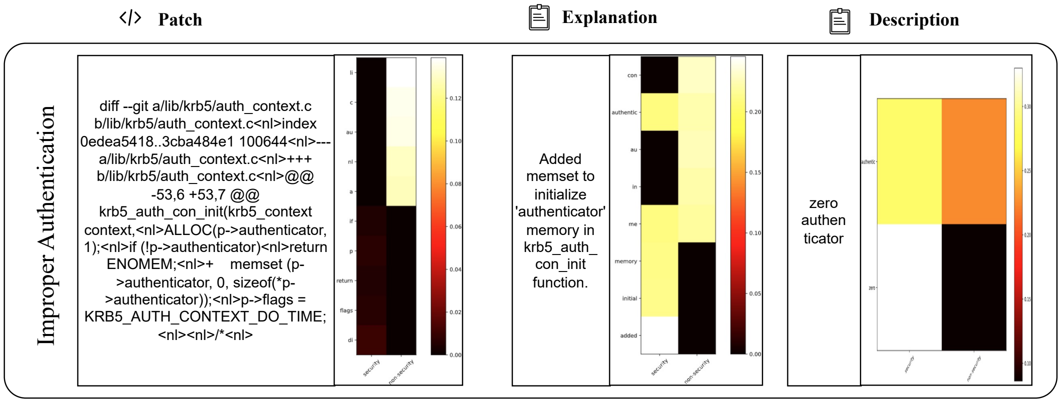


```
diff --git a/sgminer.c
b/sgminer.c<nl>index
a7dd3ab3..08697cd0 100644<nl>---
a/sgminer.c<nl>+++
b/sgminer.c<nl>@@ -518,7 +518,7
@@ struct pool *add_pool(void)
<nl>sprintf(buf, \"Pool %d\", pool-
>pool_no);<nl>pool->poolname =
strdup(buf);<nl><nl>-\tpools =
realloc(pools, sizeof(struct pool *) *
(total_pools + 2));<nl>+\tpools = (struct
pool **)realloc(pools, sizeof(struct pool
*) * (total_pools + 2));
<nl>pools[total_pools++] = pool;
<nl>mutex_init(&pool->pool_lock);
<nl>if
(unlikely(pthread_cond_init(&pool-
>cr_cond, NULL)))<nl>
```

Modified sgminer.c, adjusted realloc usage with explicit type casting.

Fixed missing realloc removed by mistake.

Figure 5: Case study in "Buffer Overflow"

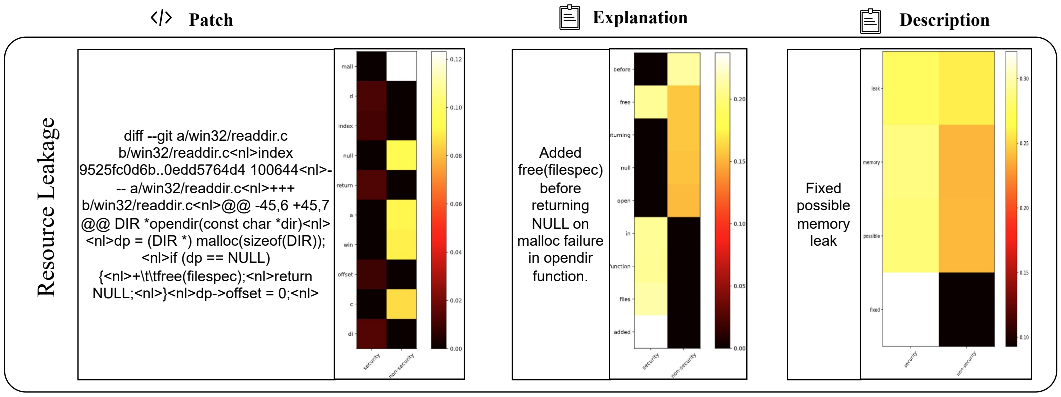


```
diff --git a/lib/krb5/auth_context.c
b/lib/krb5/auth_context.c<nl>index
0edea5418..3cba484e1 100644<nl>---
a/lib/krb5/auth_context.c<nl>+++
b/lib/krb5/auth_context.c<nl>@@
-53,6 +53,7 @@
krb5_auth_con_init(krb5_context
context,<nl>ALLOC(p->authenticator,
1);<nl>if (!p->authenticator)<nl>return
ENOMEM;<nl>+    memset (p-
>authenticator, 0, sizeof(*p-
>authenticator));<nl>p->flags =
KRB5_AUTH_CONTEXT_DO_TIME;
<nl><nl>/*<nl>
```

Added memset to initialize 'authenticator' memory in krb5_auth_con_init function.

zero authenticator

Figure 6: Case study in “Improper Authentication”

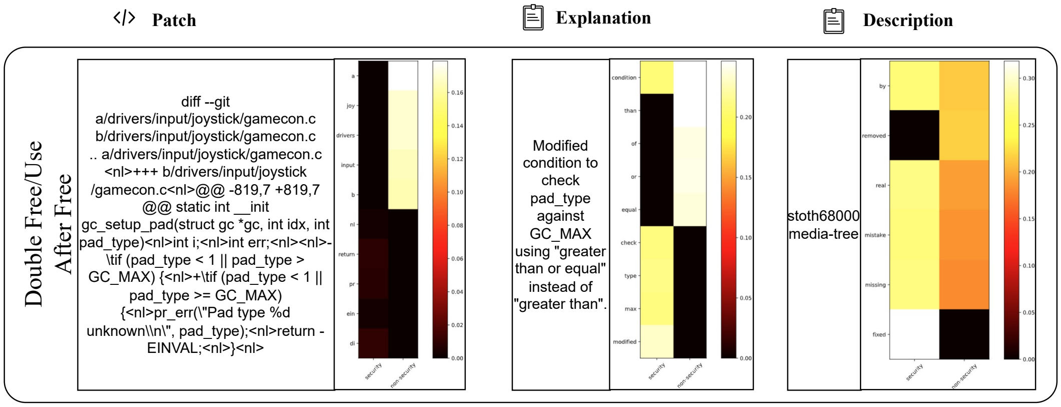


```
diff --git a/win32/readdir.c
b/win32/readdir.c<nl>index
9525fc0d6b..0edd5764d4 100644<nl>-
-- a/win32/readdir.c<nl>+++
b/win32/readdir.c<nl>@@ -45,6 +45,7
@@ DIR *opendir(const char *dir)<nl>
<nl>dp = (DIR *) malloc(sizeof(DIR));
<nl>if (dp == NULL)
{<nl>+\t\tfree(filespec);<nl>return
NULL;<nl>}<nl>dp->offset = 0;<nl>
```

Added free(filespec) before returning NULL on malloc failure in opendir function.

Fixed possible memory leak

Figure 7: Case study in “Resource Leakage”

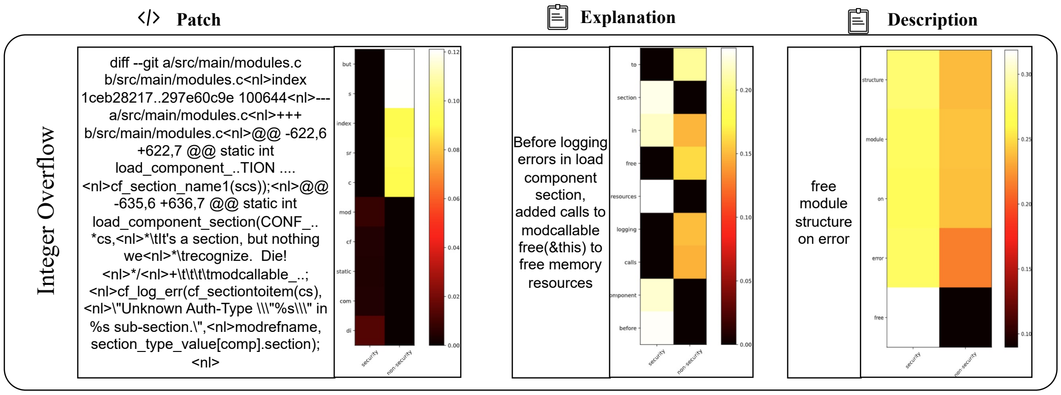


```
diff --git
a/drivers/input/joystick/gamecon.c
b/drivers/input/joystick/gamecon.c
.. a/drivers/input/joystick/gamecon.c
<nl>+++ b/drivers/input/joystick
/gamecon.c<nl>@@ -819,7 +819,7
@@ static int __init
gc_setup_pad(struct gc *gc, int idx, int
pad_type)<nl>int i;<nl>int err;<nl><nl>-
\tif (pad_type < 1 || pad_type >
GC_MAX) {<nl>+\tif (pad_type < 1 ||
pad_type >= GC_MAX)
{<nl>pr_err(\"Pad type %d
unknown\\n\", pad_type);<nl>return -
EINVAL;<nl>}<nl>
```

Modified condition to check pad_type against GC_MAX using "greater than or equal" instead of "greater than".

stoth68000 media-tree

Figure 8: Case study in “Double Free or Use After Free”

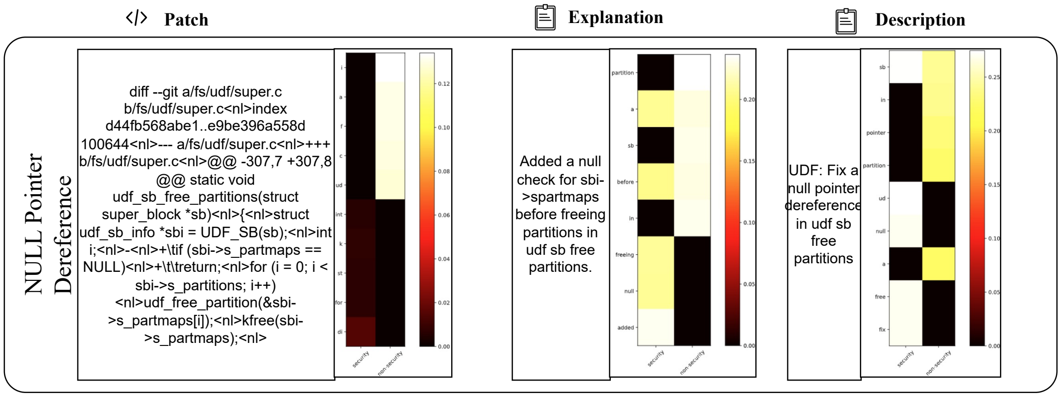


```
diff --git a/src/main/modules.c b/src/main/modules.c<nl>index 1ceb28217..297e60c9e 100644<nl>--- a/src/main/modules.c<nl>+++ b/src/main/modules.c<nl>@@ -622,6 +622,7 @@ static int load_component_..TION ....<nl>cf_section_name1(scs));<nl>@@ -635,6 +636,7 @@ static int load_component_section(CONF_.. *cs,<nl>*\tIt's a section, but nothing we<nl>*\trecognize.  Die!<nl>*/<nl>+\t\t\t\tmodcallable_..;<nl>cf_log_err(cf_sectiontoitem(cs),<nl>\"Unknown Auth-Type \\\"%s\\\" in %s sub-section.\",<nl>modrefname, section_type_value[comp].section);<nl>
```

Before logging errors in load component section, added calls to modcallable free(&this) to free memory resources

free module structure on error

Figure 9: Case study in “Integer Overflow”

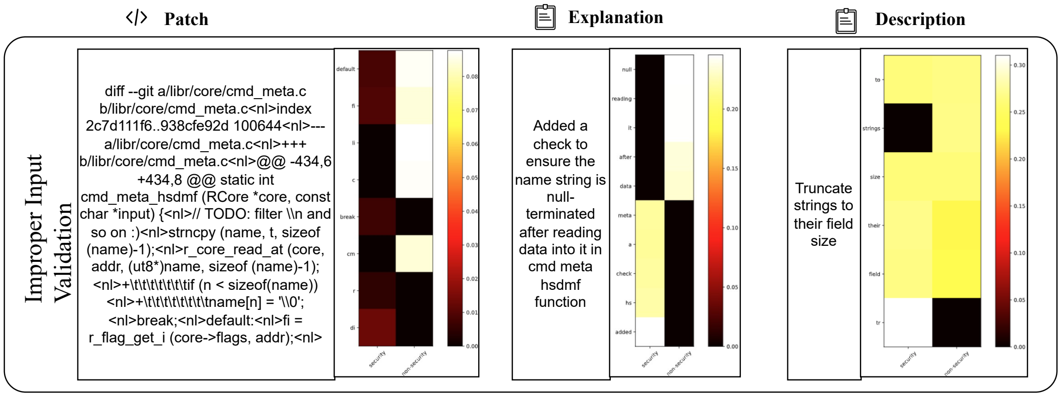


```
diff --git a/fs/udf/super.c b/fs/udf/super.c<nl>index d44fb568abe1..e9be396a558d 100644<nl>--- a/fs/udf/super.c<nl>+++ b/fs/udf/super.c<nl>@@ -307,7 +307,8 @@ static void udf_sb_free_partitions(struct super_block *sb)<nl>{<nl>struct udf_sb_info *sbi = UDF_SB(sb);<nl>int i;<nl>-<nl>+\tif (sbi->s_partmaps == NULL)<nl>+\t\treturn;<nl>for (i = 0; i < sbi->s_partitions; i++)<nl>udf_free_partition(&sbi->s_partmaps[i]);<nl>kfree(sbi->s_partmaps);<nl>
```

Added a null check for sbi->spartmaps before freeing partitions in udf sb free partitions.

UDF: Fix a null pointer dereference in udf sb free partitions

Figure 10: Case study in “NULL Pointer Dereference”

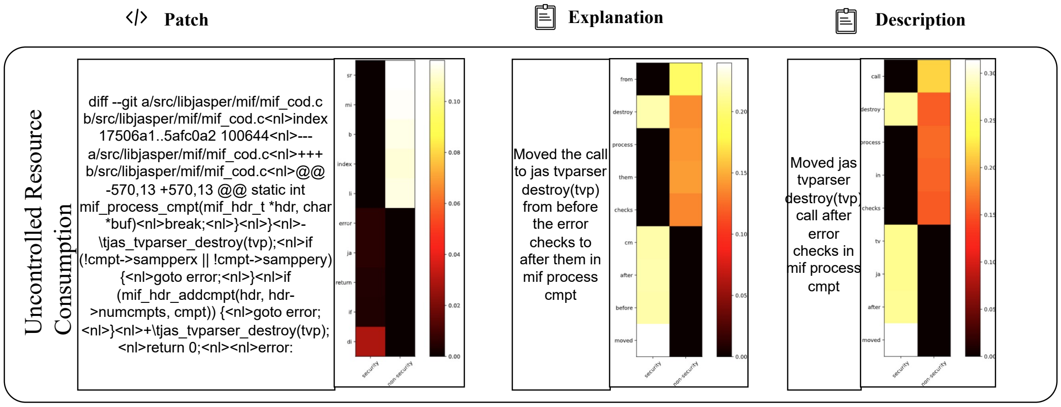


```
diff --git a/libr/core/cmd_meta.c b/libr/core/cmd_meta.c<nl>index 2c7d111f6..938cfe92d 100644<nl>--- a/libr/core/cmd_meta.c<nl>+++ b/libr/core/cmd_meta.c<nl>@@ -434,6 +434,8 @@ static int cmd_meta_hsdmf (RCore *core, const char *input) {<nl>// TODO: filter \\n and so on :)<nl>strncpy (name, t, sizeof (name)-1);<nl>r_core_read_at (core, addr, (ut8*)name, sizeof (name)-1);<nl>+\t\t\t\t\t\t\tif (n < sizeof(name))<nl>+\t\t\t\t\t\t\t\tname[n] = '\\0';<nl>break;<nl>default:<nl>fi = r_flag_get_i (core->flags, addr);<nl>
```

Added a check to ensure the name string is null-terminated after reading data into it in cmd meta hsdmf function

Truncate strings to their field size

Figure 11: Case study in “Improper Input Validation”

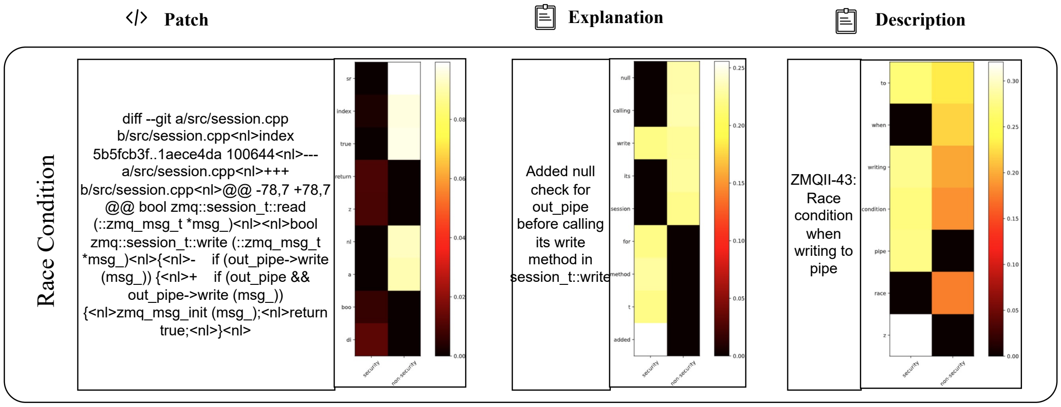


```
diff --git a/src/libjasper/mif/mif_cod.c
b/src/libjasper/mif/mif_cod.c<nl>index
17506a1..5afc0a2 100644<nl>---
a/src/libjasper/mif/mif_cod.c<nl>+++
b/src/libjasper/mif/mif_cod.c<nl>@@
-570,13 +570,13 @@ static int
mif_process_cmpt(mif_hdr_t *hdr, char
*buf)<nl>break;<nl>}<nl>}<nl>-
\tjas_tvparser_destroy(tvp);<nl>if
(!cmpt->sampperx || !cmpt->samppery)
{<nl>goto error;<nl>}<nl>if
(mif_hdr_addcmpt(hdr, hdr-
>numcmpts, cmpt)) {<nl>goto error;
<nl>}<nl>+\tjas_tvparser_destroy(tvp);
<nl>return 0;<nl><nl>error:
```

Moved the call to jas tvparser destroy(tvp) from before the error checks to after them in mif process cmpt

Moved jas tvparser destroy(tvp) call after error checks in mif process cmpt

Figure 12: Case study in “Uncontrolled Resource Consumption”

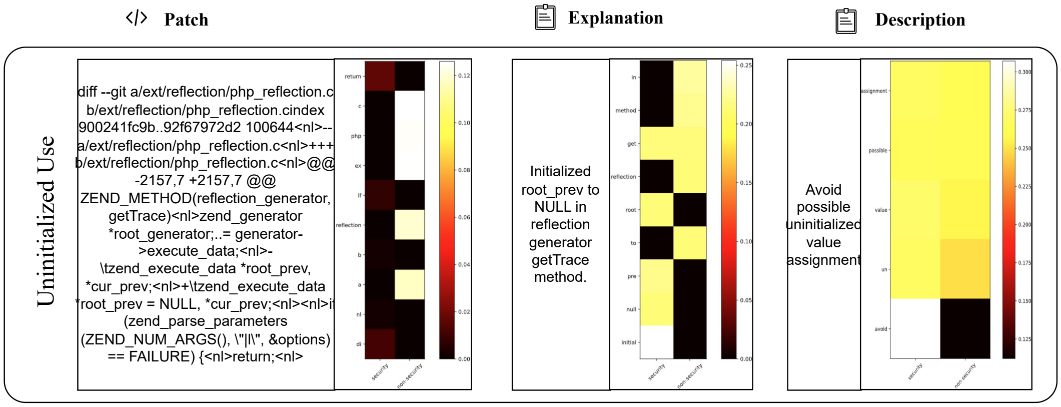


```
diff --git a/src/session.cpp
b/src/session.cpp<nl>index
5b5fcb3f..1aece4da 100644<nl>---
a/src/session.cpp<nl>+++
b/src/session.cpp<nl>@@ -78,7 +78,7
@@ bool zmq::session_t::read
(::zmq_msg_t *msg_)<nl><nl>bool
zmq::session_t::write (::zmq_msg_t
*msg_)<nl>{<nl>-    if (out_pipe->write
(msg_)) {<nl>+    if (out_pipe &&
out_pipe->write (msg_))
{<nl>zmq_msg_init (msg_);<nl>return
true;<nl>}<nl>
```

Added null check for out_pipe before calling its write method in session_t::write

ZMQII-43: Race condition when writing to pipe

Figure 13: Case study in “Race Condition”

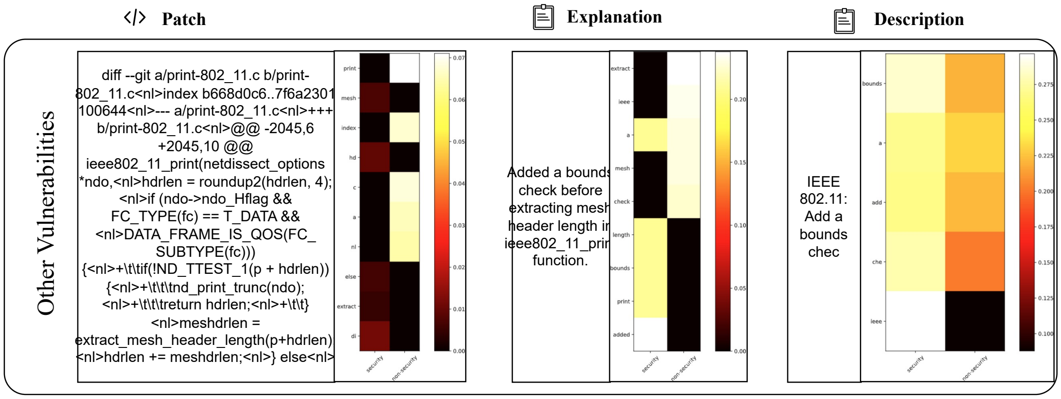


```
diff --git a/ext/reflection/php_reflection.c
b/ext/reflection/php_reflection.cindex
900241fc9b..92f67972d2 100644<nl>--
a/ext/reflection/php_reflection.c<nl>+++
b/ext/reflection/php_reflection.c<nl>@@
-2157,7 +2157,7 @@
ZEND_METHOD(reflection_generator,
getTrace)<nl>zend_generator
*root_generator;..= generator-
>execute_data;<nl>-
\tzend_execute_data *root_prev,
*cur_prev;<nl>+\tzend_execute_data
*root_prev = NULL, *cur_prev;<nl><nl>if
(zend_parse_parameters
(ZEND_NUM_ARGS(), \"|l\", &options)
== FAILURE) {<nl>return;<nl>
```

Initialized root_prev to NULL in reflection generator getTrace method.

Avoid possible uninitialized value assignment

Figure 14: Case study in “Uninitialized Use”



```
diff --git a/print-802_11.c b/print-
802_11.c<nl>index b668d0c6..7f6a2301
100644<nl>--- a/print-802_11.c<nl>+++
b/print-802_11.c<nl>@@ -2045,6
+2045,10 @@
ieee802_11_print(netdissect_options
*ndo,<nl>hdrlen = roundup2(hdrlen, 4);
<nl>if (ndo->ndo_Hflag &&
FC_TYPE(fc) == T_DATA &&
<nl>DATA_FRAME_IS_QOS(FC_
SUBTYPE(fc)))
{<nl>+\t\tif(!ND_TTEST_1(p + hdrlen))
{<nl>+\t\t\tnd_print_trunc(ndo);
<nl>+\t\t\treturn hdrlen;<nl>+\t\t}
<nl>meshdrlen =
extract_mesh_header_length(p+hdrlen)
<nl>hdrlen += meshdrlen;<nl>} else<nl>
```

Added a bounds check before extracting mesh header length in ieee802_11_prin function.

IEEE 802.11: Add a bounds chec

Figure 15: Case study in “Other Vulnerabilities”