sciendo International Journal of Mathematics and Computer in Engineering 3(1) (2025) 47–56

International Journal of Mathematics and Computer in Engineering

https://sciendo.com/journal/IJMCE

Original Study

SecuGuard: Leveraging pattern-exploiting training in language models for advanced software vulnerability detection

Mahmoud Basharat[1], Marwan Omar[2†]

[1]Capitol Technology University, Maryland, 20708, USA

[2]Illinois Institute of Technology, Warrenville, Illinois, 60616, USA

Communicated by Haci Mehmet Baskonus; Received: 28.10.2023; Accepted: 16.01.2024; Online: 02.06.2024

Abstract

Identifying vulnerabilities within source code remains paramount in assuring software quality and security. This study introduces a refined semi-supervised learning methodology that capitalizes on pattern-exploiting training coupled with cloze-style interrogation techniques. The research strategy employed involves the training of a linguistic model on the Software Assurance Reference Dataset (SARD) and Devign datasets, which are replete with vulnerable code fragments. The training procedure entails obscuring specific segments of the code and subsequently prompting the model to ascertain the obfuscated tokens. Empirical analyses underscore the efficacy of our method in pinpointing vulnerabilities in source code, benefiting substantially from patterns discerned within the code fragments. This investigation underscores the potential of integrating pattern-exploiting training and cloze-based queries to enhance the precision of vulnerability detection within source code.

Keywords: Language models, software vulnerabilities, vulnerability detection, cloze-style questions, pattern-exploiting training, RoBERTa .
AMS 2020 codes: 68T07.

1 Introduction

The digital platform domain is witnessing an escalation in intricate and malevolent cyber incursions. These transgressions predominantly harness system susceptibilities, defined as system lacunae manipulable by cyber adversaries for multifaceted gains [1–34]. A significant precipitant of these cyber onslaughts is the inherent software vulnerabilities. Even with significant strides by academic and industrial entities in fortifying software integrity, the continued emergence of vulnerabilities, as underscored by annual records in the Common Vulnerabilities and Exposures (CVE) database [16], remains alarming [28]. Considering the inevitable nature of these susceptibilities, their early detection becomes imperative. Static analysis of source code offers one such detection avenue, embracing methodologies ranging from code similarity assessment to pattern-recognition techniques. Notably, while code similarity evaluation can pinpoint vulnerabilities emanating from code replication,

†Corresponding author.
Email address: drmarwan.omar@gmail.com

sciendo
ISSN 2956-7068 doi:10.2478/ijmce-2025-0005
Open Access.© 2025 Omar et al., published by Sciendo.
This work is licensed under the Creative Commons Attribution alone 4.0 License.

considerable false negatives may incur [1, 2, 9, 10, 15–24]. In a bid to tackle these vulnerability detection quandaries, the academic sphere has introduced methods such as fuzzing, symbolic scrutiny, and rule-centric testing. These methodologies, despite their acclaim, are hindered by constraints like manual attack signature and pattern definitions, compromising their efficacy on extensive code repositories. Moreover, conventional techniques for vulnerability detection are plagued by false positives, performance hindrances, and challenges in vulnerability typology discernment [4, 18]. In a progressive move, the integration of machine learning-specifically deep learning-into vulnerability detection paradigms has been pursued. Such integrations streamline manual interventions and expedite the vulnerability detection process. Forefront machine learning algorithms like longshort-term memories (LSTMs) and transformers undertake the classification of API sequences from program execution traces into benign or malignant categories, even prognosticating the exploit genre. However, their computational voracity diminishes their applicability [22]. This research endeavors to harness a "pattern-exploiting training" (PET) and "iterative pattern-exploiting training" (iPET) paradigm, leveraging cloze-style interrogation, to architect an expansive linguistic model aimed at software vulnerability detection. In this context, cloze interrogatives, which entail blanks within content that necessitate completion [26], are framed upon code fragments; the lacunae represent the extant vulnerabilities. The rationale for our methodology is grounded in the notion that a comprehensive language model, cultivated on a vast dataset of code-based cloze interrogatives, imbibes both the vulnerable and benign code exemplars. Such exposure equips the model with the proficiency to discern code configurations suggestive of vulnerabilities. This trained model can then discern potential vulnerabilities in fresh code segments by complementing the lacunae in the cloze interrogatives. For instance, code manifesting a probable buffer overflow vulnerability is analyzed using the PET paradigm by determining the code configurations associated with buffer overflow susceptibilities and subsequently generating cloze interrogatives for linguistic model training. Following this, the linguistic model, equipped with cloze training, becomes adept at examining new code segments, pinpointing configurations reminiscent of established vulnerability blueprints. This modus operandi avails an automated vulnerability pattern detection, obviating the necessity for expert manual examination. In this scholarly endeavor, we introduce "VulDefend," a vulnerability detection architecture employing the RoBERTa model for C and C++ source codes. Our seminal contributions encompass:

1. The conceptualization of VulDefend, an innovative system harnessing pattern-exploiting training and cloze methodology for software vulnerability detection, capitalizing on expansive linguistic model competencies.

2. Through benchmark datasets and the RoBERTa-based linguistic model, we evidence VulDefend's proficiency in identifying code susceptibilities across multiple programming languages, inclusive of C/C++ and Java.

3. Comparative analyses depict VulDefend's superior performance over dual contemporary benchmark techniques in software vulnerability identification.

2 Related work

The quest to identify vulnerabilities in source code has engrossed researchers in recent years. Myriad studies have unveiled methodologies that employ machine learning techniques for this pursuit. Some concentrate on static analysis, extracting salient features from the code to be input into predictive machine-learning frameworks [1, 2], while others hinge on dynamic analysis, running the code and monitoring its behavior to discern vulnerabilities [3,4]. A nascent inclination towards harnessing deep learning models for source code vulnerability detection has been observed. Certain researches harness recurrent neural networks (RNNs) to encapsulate the code, whether in its pristine form [5, 6] or post its transmutation into an abstract syntax tree (AST) [7, 8]. Meanwhile, other investigations employ transformers, which have gained acclaim in the realm of natural language processing [9, 10]. Literature has extensively analyzed deep learning architectures like Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs) for vulnerability detection [3, 12–14, 25, 29, 31, 34]. Yet, these models necessitate structured data to discern vulnerability-associated features. This spurred the inception

of techniques, including lexed C/C++ code representation [13], code gadgets [25], code property graphs [35], augmented code gadgets considering code attention and system dependency graphs, and minimal intermediate representation learning [31]. Software vulnerability detection has also seen the application of graph neural networks [34]. The Devign model embodies compound programmatic representations like abstract syntax trees, control flow, and data flow from the source code. In an archetype, Russell et al. [27] showcased the prowess of deep learning in identifying software vulnerabilities within raw source code. Their innovative proposition utilized CNN and RNN as feature extractors that subsequently informed a Random Forest classifier. This paradigm, when trialed on pragmatic datasets, rendered an impressive AUC score of 90.4 Building upon the momentum, Vuldeepecker [36] emerged, offering multi-class vulnerability classification and even specifying the vulnerability location within the source code. Research has also navigated graph-centric vulnerability detection strategies. Notably, Devign [35] and DeepWukong [7] employed Graph Neural Network models. Beyond traditional languages, DeepTective [25] discerned vulnerabilities in PHP, and another study [32] probed HTML5-based applications. Dataset quality enhancements for deep learning-based vulnerability detection have also been prioritized, as seen with REVEAL [6] and D2A. In a recent endeavor, Naif et al. [11] launched VulBERTa, a deeply representational model of C/C++ code. However, it lacked a mechanism to identify novel 0-day vulnerabilities within real-world, open-source ventures. Our research dovetails with the contemporary thrust towards leveraging deep learning for source code vulnerability detection. Distinctively, we employ pattern-exploiting training combined with cloze queries to endow a compact student model with the insights of a larger language model. The ultimate aim is to amplify the student model's vulnerability detection prowess. This strategy delineates itself from prior work which primarily revolved around using RNNs, transformers, or knowledge distillation with deep learning for varied tasks.

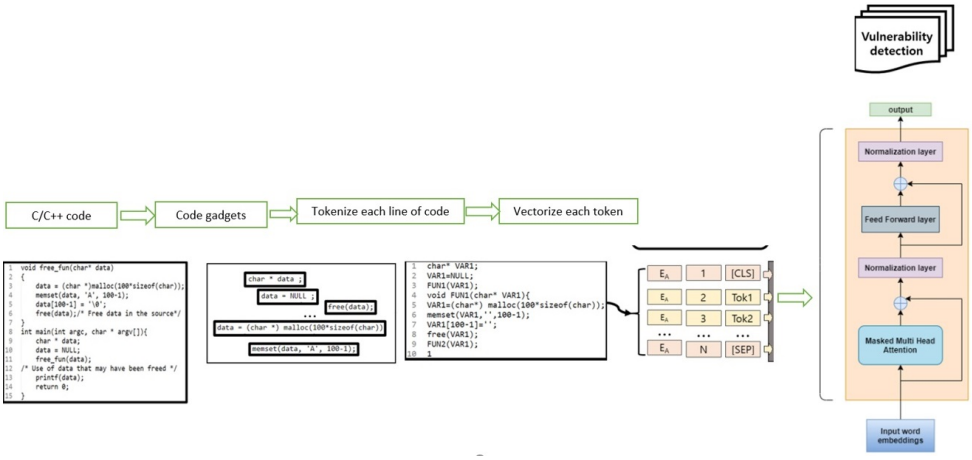

Fig. 1 An overview of our defense framework.

The underlying rationale for employing pattern-exploiting training to reshape input data into cloze-style queries in a semi supervised learning environment revolves around manipulating discernible patterns within input data. The objective is to generate fresh training instances [26], subsequently honing machine learning models specific to vulnerability detection. This methodology gains prominence particularly in scenarios with sparse labelled data, granting the model the capacity to discern from inherent patterns rather than being excessively dependent on labelled instances. Let's delve into a hypothetical scenario: tasked with the detection of vulnerabilities within function calls utilizing a benchmark dataset, for instance, REVEAL [6]. Contextually, within source code function calls, core patterns would encompass function names invoked, argument types passed to these functions, and the associated security ramifications of such calls. Morphing these samples into cloze-style questions, where portions of this data are concealed, empowers the model to prognosticate the concealed data.

Deploying this technique to the REVEAL dataset, aimed at vulnerability detection within source code, yields multifaceted advantages. Primarily, the emphasis on inherent data patterns equips the model with a broader and

more generalized data representation, subsequently applicable to novel code fragments. The semi-supervised learning milieu further capacitates the model to exploit data patterns, even in the face of limited labelled data. Furthermore, cloze-style queries offer the model a more regimented learning platform, potentially augmenting its efficacy. To encapsulate, harnessing pattern-exploiting training to metamorphose input data into cloze-style queries within a semi-supervised learning context emerges as a propitious avenue for unearthing vulnerabilities in source code. By tapping into inherent data patterns, the model can extrapolate concealed data and spotlight potential security predicaments in unfamiliar code snippets. Such an innovative strategy holds the promise of bolstering software security, allowing for early vulnerability interception during the development trajectory.

3 Methodology

This section delineates the intricacies of the methodology incorporated for discerning vulnerabilities within source code, leveraging PET and cloze-style queries. We will unpack the preprocessing regime, elucidate the training paradigm, and touch upon the evaluative measures to gauge model performance. The overarching aim is to furnish readers with a lucid grasp of the executional steps and the rationale guiding these choices. The initial preprocessing phase is pivotal, morphing raw source code into a digestible format suitable for PET. The flowchart in Figure 1 captures this transformation succinctly. The processes entail:

Tokenization: Dissecting source code into discrete tokens, categorizing them as keywords, variables, or punctuation. Normalization: Homogenizing these tokens to maintain data uniformity. A case in point: converting all function names to lowercase.

Labeling: Assigning a descriptive tag to each token to indicate its role within the code. This might entail classifying tokens as function names, variables, or keywords.

Cloze generation: Exploiting these processed tokens to spawn cloze-style questions, wherein a code segment is substituted with a mask token, prompting the model to forecast the concealed segment.

These steps are instrumental in curating a structured dataset, primed for training, ensuring the language model adeptly comprehends source code intricacies vital for vulnerability detection.

3.1 Pattern-exploiting training

Visualize a training set D comprising N input-output duos, wherein each input x_i symbolizes a code fragment and its counterpart output y_i denotes the vulnerability status of said code. The ambition is to harness PET, metamorphosing these inputs into cloze-styled statements for vulnerability detection.

This transformation begins with discerning prominent patterns P within the input. Each pivotal pattern p_j correlates with an index set I_j, ensuring that for every $i \in I_j, x_i$ encapsulates p_j. This leads to a new input set X, formulated by masking segments of each input x_i such that $X_i = \text{cloze}(x_i, p_j)$ for some $p_j \in P$. For clarity, consider the pattern "function call" and an input x_i with the function call "strcpy". The transformed $X_i' = \text{cloze}(x_i, p_j)$ would mask "strcpy". A machine learning model f is then calibrated using the altered inputs X_i' and associated outputs y. Predictions for new inputs utilize the same cloze transformation. Mathematically, the formulae for this transformation and subsequent prediction are:

$$\begin{aligned} X &= \text{cloze}(x_i, p_j) \ \text{ for } \ i \in 1, 2, \cdots, N, p_j \in P \\ f &= \text{train}(X, y) \\ y_{new} &= f(x_{new}). \end{aligned}$$

Cross-entropy loss, a staple in supervised learning, gauges discrepancies between forecasted and actual labels. Formally:

$$L(p, q) = -\Sigma_{k=1}^{K} q_k \log(p_k).$$

Minimizing this loss during training fine-tunes the model, ensuring predictions mirror true probabilities. Post-training, this model becomes adept at vulnerability detection using cloze-style queries.

Leveraging the BERT architecture, one could optimize a pre-existing BERT model using reformulated inputs X' and outputs y, aiming to predict concealed segments in cloze-style queries.

4 System overview

Within this segment, we introduce "VulDefend" a classification model sculpted atop the colossal language model BERT, envisaged to autonomously unearth security vulnerabilities in source code, as portrayed in Figure 1. This fine-tuned BERT incarnation deciphers vectors tied to vulnerable code components drawn from a targeted source. VulDefend ingests inputs as elongated character strings, representing C files. A subsequent tokenizer dissects this string into words and sub-words, noting that syntactic characters (e.g., periods, semicolons) are treated as standalone words. Following tokenization, an encoder translates these words into vectorial form, either ingesting them piecemeal into the model or in bulk. In our vulnerability identification construct, the output vector mirrors the count of vulnerability classes within the dataset. Given a dataset with 124 unique vulnerability classes, the resultant output vector boasts a 124-dimensional span. A subsequent Softmax function refines this vector into a probability distribution, ensuring all probabilities cumulatively equate to 1. Each vector component forecasts the likelihood of its associated vulnerability class manifesting within the analyzed code file.

4.1 Data sources

In our research, we have selected multiple datasets which are considered benchmarks in the realm of vulnerability detection. These datasets, previously employed in several studies such as SARD [34], D2A [33], REVEAL [6], and the Devign dataset [30].

1. *Devign:* Introduced in [30], the Devign dataset stands as a practical representation of data tailored for vulnerability detection in software. It's an aggregation of C/C++ source code functions pulled from renowned open-source projects such as QEMU and FFmpeg. The data's reliability is reinforced by a rigorous two-phase validation process performed by seasoned security experts.

2. *SARD:* Offered as a robust reference, the SARD [34] acts as a repository of details and tools catering to software assurance and its security. With a focus on real-world vulnerabilities and the code that accompanies them, SARD's library spans a range of vulnerability types including buffer overflows and XSS. Updated consistently, SARD is recognized in both academic and industrial circles for assessing the efficacy of software assurance tools.

3. *REVEAL:* The REVEAL dataset, highlighted in [6], emerges as an answer to the redundancy and skewed vulnerability distribution in current datasets. By concentrating on binary detection, REVEAL incorporates source code from notable open-source endeavors, namely the Linux Debian kernel and Chromium. This dataset stands as a fresh portrayal of authentic vulnerability situations and fosters further exploration in this domain.

4. *D2A:* Originated by the IBM Research division [33], the D2A dataset stands as an exhaustive reservoir of real world vulnerability detection data. Incorporating code from renowned open-source projects such as FFmpeg, Nginx, and more, it's structured using a differential analysis technique that labels issues pinpointed by static analysis instruments.

4.2 Experimental setup

Our experiments are conducted on an ASUS TUF Gaming laptop powered by an Intel Core i7-8th gen processor. The processor boasts six cores, each having a peak operational speed of 2.2 GHz.

Table 1 The average accuracy and the standard deviation for BERT base on SARD, D2A, REVEAL and DEVIGN over 5 training set sizes.

Line	Examples	Method	SARD	D2A	REVEAL	Devign
1.	$\mid T \mid = 0$	unsupervised (avg)	38.8 ±9.6	69.5 ±7.2	44.0 ±9.1	39.1 ±4.3
2.	$\mid T \mid = 0$	unsupervised (max)	42.8 ±0.0	79.4 ±0.0	56.4 ±0.0	43.8 ±0.0
3.	$\mid T \mid = 0$	iPet	66.7 ±0.2	89.5 ±0.1	73.7 ±0.1	63.6 ±0.1
4.	$\mid T \mid = 15$	supervised	32.1 ±1.6	25.0 ±0.1	10.1 ±0.1	34.2 ±2.1
5.	$\mid T \mid = 15$	Pet	52.9 ±0.1	87.5 ±0.0	63.8 ±0.2	41.8 ±0.1
6.	$\mid T \mid = 15$	iPet	57.6 ±0.0	89.3 ±0.1	70.7 ±0.1	43.2 ±0.0
7.	$\mid T \mid = 60$	supervised	44.8 ±2.7	82.1 ±2.5	52.5 ±3.1	45.6 ±1.8
8.	$\mid T \mid = 60$	Pet	60.0 ±0.1	86.3 ±0.0	66.2 ±0.1	63.9 ±0.0
9.	$\mid T \mid = 60$	iPet	64.7 ±0.1	88.4 ±0.1	69.7 ±0.0	67.4 ±0.3
10.	$\mid T \mid = 200$	supervised	53.0 ±3.1	86.0 ±0.7	62.9 ±0.9	47.9 ±2.8
11.	$\mid T \mid = 200$	Pet	61.9 ±0.0	88.3 ±0.1	69.2 ±0.0	74.7 ±0.3
12.	$\mid T \mid = 200$	iPet	62.9 ±0.0	89.6 ±0.1	71.2 ±0.1	78.4 ±0.7
13.	$\mid \tau \mid = 1000$	supervised	63.0 ±0.5	86.9 ±0.4	70.5 ±0.3	73.1 ±0.2
14.	$\mid \tau \mid = 1000$	Pet	68.8 ±0.1	89.9 ±0.2	72.7 ±0.0	85.3 ±0.2

5 Outcome analysis

In our research, we present the outcome of a vulnerability detection exercise as shown in Table 1. For the three distinct training sessions, both mean accuracy and its standard deviation are detailed. The initial two rows (L1-L2) in the table showcase unsupervised method outcomes, highlighting the highest average across all test datasets. This prominent variation between the two rows underscores the necessity to consult the test set to gauge the best-performing method. Zeroshot iPET consistently surpasses the unsupervised standards across datasets and even outdoes regular supervised training with 1000 samples for D2A. On the other hand, with only 15 samples, conventional supervised learning is no better than a random guess. Yet, with successive generations of iPET training, PET consistently shows improvements. Although the advantage of PET and iPET narrows as we augment the training dataset, even with 60 and 100 samples, PET remains substantially superior.

Table 2 A comparison of PET with VulBERTa and VulDeBERT methods using BERT (base).

Ex.	Method	SARD	D2A	REVEAL	Devign
$\mid T \mid = 15$	VulDeBERT	40.45	72.6	36.7	34.7
$\mid T \mid = 15$	VulBERTa	43.23	81.1	320.6	32.9
$\mid T \mid = 15$	Pet	49.60	84.1	59.0	39.5
$\mid T \mid = 15$	iPet	54.60	87.5	67.0	42.1
$\mid T \mid = 60$	VulDeBERT	46.6	83.0	60.2	40.8
$\mid T \mid = 60$	VulBERTa	39.5	84.8	61.5	34.8
$\mid T \mid = 60$	Pet	55.3	86.4	63.3	55.1
$\mid T \mid = 60$	iPet	57.7	87.3	69.6	56.3

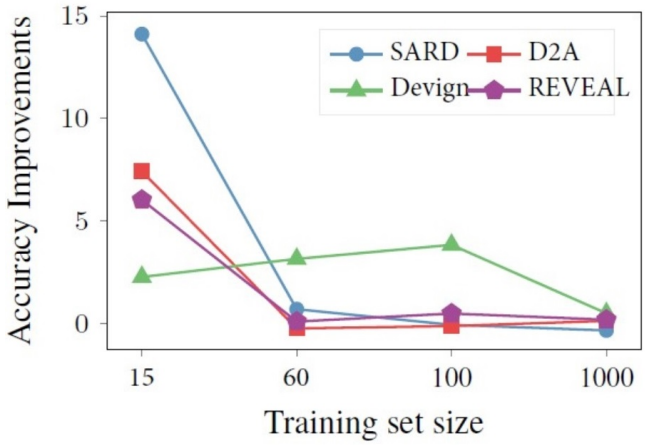

Fig. 2 The inclusion of Additional Language Modeling during training resulted in improvements in accuracy for PET.

5.1 Enhanced language modeling

We explored the influence of an additional language modeling task on PET's performance. Figure 2 showcases the enhanced outcomes from this task over four distinct training dataset sizes. The outcomes suggest a significant boost from the supplementary task with a mere 15 samples. But as training data grows, this task's relevance lessens, occasionally even diminishing performance. Nevertheless, for the Devign dataset, this task consistently enhances outcomes.

5.2 iPET explanation

"Iterative distillation and amplification", or iPET, builds on the foundational PET algorithm. It augments PET's capabilities by cyclically transferring insights from the primary model to a secondary model, intensifying their predictive differences. This cycle helps the secondary model absorb intricate nuances from the primary model, leading to enhanced outcomes [5, 8]. In our method, merging insights from individual models might curtail mutual learning since some patterns may falter compared to others. This results in numerous mislabelled samples in the ultimate dataset. We combat this through the iterative iPET strategy. Its essence is in training multiple model generations on expanding datasets. We enlarge the foundational dataset T by integrating labeled samples from D using some trained PET models. Successive PET model generations are then trained, allowing consistent refinement towards a robust final model.

5.3 Comparative study

This study's pivotal segment is comparing PET with leading software vulnerability detection tools, namely, VulBERTa and VulDeBERT. PET operates on pattern identification, while the latter two demand supervised training using extensive labeled samples on a 12-layer Transformer, i.e., BERT (base). We contrasted several iterations of both methods directly on the test dataset and highlighted the peak performances. Table 2 clearly indicates the marked superiority of both PET and iPET across all datasets when juxtaposed with the other two. This distinction clearly showcases the merit of integrating knowledge distillation. A glance at Table 2 reinforces that PET and iPET outshine VulBERTa and VulDeBERT, emphasizing the value of amalgamating knowledge distillation and transfer learning methodologies.

5.4 VulDefend's identified vulnerabilities

We exhibit some vulnerability instances detected via our methodology, followed by a performance discourse on varied vulnerability classes. Vulnerabilities identified by our method:

SQL injection risk: Several code snippets harbored SQL injection risks, identified by our method. Our model spotted patterns that let unchecked user input feed directly into SQL statements sans validation.

Cross-site scripting (XSS) risk: Similarly, our method pinpointed multiple XSS risks. Patterns allowing unchecked user inputs to populate web content without validation were identified by our model. Our method adeptly spotted vulnerabilities with evident code patterns, like SQL injection and XSS risks. But, the efficacy dipped for intricate vulnerabilities involving multitiered code interactions. Examples include race conditions or privilege escalation, which might lack distinct patterns. To summarize, our method shows potential in identifying select vulnerability categories, especially those with clear patterns. Still, a broader vulnerability scope requires evaluation, and strategies to spot intricate vulnerabilities warrant further investigation.

6 Challenges faced

The technique of utilizing cloze-style patterns in Pattern Exploiting Training (PET) alongside language models for spotting vulnerabilities shows potential, but it isn't without challenges. First, data constraints: Gathering labeled data for cloze-style training can be labor-intensive and challenging. The model's effectiveness is bound to the quality and volume of this data. Second, versatility challenges: Although a model fine-tuned for one coding language might excel, transitioning to different languages or varied syntax might pose hurdles. Third, inherent model constraints: Language models, despite their vast training, might not grasp coding intricacies. They might falter with highly sophisticated or unique code features. Fourth, domain-specific issues: Recognizing vulnerabilities requires comprehensive code and security concept comprehension, which cloze-training might not always capture comprehensively. Lastly, understanding challenges: Deciphering the reasoning behind neural network-based model predictions can be daunting. To mitigate these challenges, we're exploring innovative strategies like parameter-sharing and experimenting with diverse language models like reformers to refine our vulnerability detection mechanisms

7 Versatility and evolution

For the PET methodology to cater to various coding languages, model architecture and training cloze questions might need adjustments. An approach could be employing pre-training datasets specific to a language. For instance, a Python specific dataset might feature prominent Python libraries, while a JavaScript one would highlight key web development libraries. Future endeavors involve refining the PET technique for enhanced accuracy in spotting vulnerabilities across coding languages. This includes innovating model structures, perfecting the training regime, and weaving the model into existing software creation processes. Efforts are also directed at expanding the model's testing parameters, pinpointing false positive and negative origins, and strategizing the integration of human expertise in the vulnerability detection chain.

8 Conclusion

The cybersecurity domain critically hinges on proficient, efficient, and precise language models, especially when preemptively pinpointing software vulnerabilities to bolster security frameworks. Our empirical analysis confirms the merit of using cloze-style patterns in PET with language models for scrutinizing source code for vulnerabilities. Transforming input samples into cloze-style queries enables the model to forecast code gaps and, consequently, detect vulnerabilities. The potential of this approach in enhancing the precision and efficiency of source code vulnerability detection is evident. Paving the way forward, we aspire to innovate around parameter-sharing concepts and harness avant-garde language models like GPT and reformers to architect sophisticated and resilient models for software vulnerability detection.

9 Declarations

9.1 Conflict of interests:

The authors hereby declare that there is no conflict of interests regarding the publication of this paper.

9.2 Funding:

There is no funding regarding the publication of this paper.

9.3 Author's contributions:

M.B.-Conceptualization, Methodology, Validation, Formal Analysis. M.O.-Investigation, Resources, Data Curation, Writing-Original Draft, Writing-Review and Editing. The authors have worked equally when writing this paper. All authors read and approved the final submitted version of this manuscript.

9.4 Acknowledgement:

Many thanks to the reviewers for their constructive comments on revisions to the article. The research is partially supported by NSFC (NO: 12161094).

9.5 Data availability statement:

This paper is a theoretical work and has been developed with data which is still being used for development and research.

9.6 Using of AI tools:

The authors declare that they have not used Artificial Intelligence (AI) tools in the creation of this article.

References

[1] Abbasi R., Bashir A.K., Mateen A., Amin F., Ge Y., Omar M., Efficient security and privacy of lossless secure communication for sensor-based urban cities, IEEE Sensors Journal, DOI: 10.1109/JSEN.2023.3305716, 2024.
[2] Kinoon M.A., Omar M., Mohaisen M., Mohaisen D., Security breaches in the healthcare domain: a spatiotemporal analysis, Computational Data and Social Networks: 10th International Conference, CSoNet 2021, Virtual Event, 15–17 November 2021, 171–183, 2021.
[3] Alharbi A.R., Hijji M., Aljaedi A., Enhancing topic clustering for Arabic security news based on k-means and topic modelling, IET Networks, 10(6), 278–294, 2021.
[4] Aluru S.S., Mathew B., Saha P., Mukherjee A., Deep learning models for multilingual hate speech detection, arXiv:2004.06465, 2020.
[5] Beyer L., Zhai X., Royer A., Markeeva L., Anil R., Kolesnikov A., Knowledge distillation: a good teacher is patient and consistent, 2022 IEEE/CVF Conference on Computer Vision and Pattern Recognition, 18-24 June 2022, New Orleans, Los Angeles, USA, 10925–10934, 2022.
[6] Chakraborty S., Krishna R., Ding Y., Ray B., Deep learning based vulnerability detection: Are we there yet?, IEEE Transactions on Software Engineering, 48(9), 3280–3296, 2021.
[7] Cheng X., Wang H., Hua J., Xu G., Sui Y., DeepWukong: statically detecting software vulnerabilities using deep graph neural network, ACM Transactions on Software Engineering and Methodology, 30(3), 1–33, 2021.
[8] Furlanello T., Lipton Z., Tschannen M., Itti L., Anandkumar A., Born again neural networks, International Conference on Machine Learning, PLMR, 1607–1616, 2018.
[9] Gholami S., Omar M., Can a student large language model perform as well as it's teacher?, arXiv:2310.02421, 2023.
[10] Gholami S., Omar M., Do generative large language models need billions of parameters?, arXiv:2309.06589, 2023.
[11] Hanif H., Maffeis S., VulBERTa: simplified source code pre-training for vulnerability detection, 2022 International Joint Conference on Neural Networks, 18-23 July 2022, Padua, Italy, 1–8, 2022.
[12] Kim S., Woo S., Lee H., Oh H., VUDDY: a scalable approach for vulnerable code clone discovery, 2017 IEEE Symposium on Security and Privacy, IEEE, 22-26 May 2017, San Jose, California, USA, 595–614, 2017.
[13] Kim S., Choi J., Ahmed M.E., Nepal S., Kim H., VulDeBERT: a vulnerability detection system using BERT, 2022 International Symposium on Software Reliability Engineering Workshops, IEEE, 31 October 3 November 2022, Char-

lotte, New York, USA, 69–74, 2022.
[14] Li Z., Zou D., Xu S., Jin H., Qi H., Hu J., VulPecker: an automated vulnerability detection system based on code similarity analysis, Proceedings of the 32nd Annual Conference on Computer Security Applications, Association for Computing Machinery New York USA, 5–8 December 2016, Los Angeles, California, USA, 201–213, 2016.
[15] Omar M., Application of machine learning (ML) to address cybersecurity threats, In Machine Learning for Cybersecurity: Innovative Deep Learning Solutions, Springer, 1–11, 2022.
[16] Omar M., Machine Learning for Cybersecurity: Innovative Deep Learning Solutions, Springer, 2022.
[17] Omar M., Machine Learning for Cybersecurity: Innovative Deep Learning Solutions (Chapter: Malware anomaly detection using local outlier factor technique), Springer, 2022.
[18] Omar M., Backdoor learning for NLP: recent advances, challenges, and future research directions, arXiv:2302.06801, 2023.
[19] Omar M., VulDefend: a novel technique based on pattern exploiting training for detecting software vulnerabilities using language models, 2023 IEEE Jordan International Joint Conference on Electrical Engineering and Information Technology, IEEE, 22–24 May 2023, Amman, Jordan, 287–293, 2023.
[20] Omar M., Burrell D., From text to threats: a language model approach to software vulnerability detection, International Journal of Mathematics and Computer in Engineering, 2(1), 23–34, 2024.
[21] Omar M., Choi S., Nyang D., Mohaisen D., Quantifying the performance of adversarial training on language models with distribution shifts, Proceedings of the 1st Workshop on Cybersecurity and Social Sciences, 30 May 2022, Nagasaki Japan, 3–9, 2022.
[22] Omar M., Choi S., Nyang D., Mohaisen D., Robust natural language processing: recent advances, challenges, and future directions, arXiv:2201.00768, 2022.
[23] Omar M., Jones R., Burrell D.N., Dawson M., Nobles C., Mohammed M., Bashir A.K., Harnessing the power and simplicity of decision trees to detect IoT Malware, Transformational Interventions for Business Technology and Healthcare, 215–229, 2023.
[24] Omar M., Mohaisen D., Making adversarially-trained language models forget with model retraining: a case study on hate speech detection, Companion Proceedings of the Web Conference 2022, Virtual Event, 25–29 April 2022, Lyon, France, 887–893, 2022.
[25] Rabheru R., Hanif H., Maffeis S., DeepTective: detection of PHP vulnerabilities using hybrid graph neural networks, Proceedings of the 36th Annual ACM Symposium on Applied Computing, Virtual Event, Republic of Korea, 22–26 March 2021, 1687–1690, 2021.
[26] Radford A., Wu J., Child R., Luan D., Amodei D., Sutskever I., Language models are unsupervised multitask learners, OpenAI Blog, 1(8), 9, 2019.
[27] Russell R., Kim L., Hamilton L., Lazovich T., Harer J., Ozdemir O., Ellingwood P., McConley M., Automated vulnerability detection in source code using deep representation learning, 2018 17th IEEE international conference on machine learning and applications, IEEE, 17-20 December 2018, Orlando, Florida, USA, 757–762, 2018.
[28] Saleem M.A., Li X., Mahmood K., Shamshad S., Ayub M.F., Bashir A.K., Omar M., Provably secure conditional-privacy access control protocol for intelligent customers-centric communication in VANET, IEEE Transactions on Consumer Electronics, 2023.
[29] Salimi S., Kharrazi M., VulSlicer: vulnerability detection through code slicing, Journal of Systems and Software, 193, 111450, 2022.
[30] Shoeybi M., Patwary M., Puri R., LeGresley P., Casper J., Catanzaro B., Megatron-LM: training multi-billion parameter language models using model parallelism, arXiv:1909.08053, 2019.
[31] Yamaguchi F., Golde N., Arp D., Rieck K., Modeling and discovering vulnerabilities with code property graphs, 2014 IEEE Symposium on Security and Privacy, 590–604, 2014.
[32] Yan R., Xiao X., Hu G., Peng S., Jiang Y., New deep learning method to detect code injection attacks on hybrid applications, Journal of Systems and Software, 137, 67–77, 2018.
[33] Zheng Y., Pujar S., Lewis B., Buratti L., Epstein E., Yang B., Laredo J., Morari A., Su Z., D2A: a dataset built for AI-based vulnerability detection methods using differential analysis, 2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice, IEEE, 25-28 May 2021, Madrid, Spain, 111–120, 2021.
[34] Zhou X., Verma R.M., Vulnerability detection via multimodal learning: datasets and analysis, Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security, 30 May– 3 June 2022, Nagasaki, Japan, 1225–1227, 2022.
[35] Zhou Y., Liu S., Siow J.K., Du X., Liu Y., Advances in Neural Information Processing Systems (Chapter: Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks), 10197-10207, 2019.
[36] Zou D., Wang S., Xu S., Li Z., Jin H., μVulDeePecker: a deep learning-based system for multiclass vulnerability detection, IEEE Transactions on Dependable and Secure Computing, 18(5), 2224–2236, 2021.